# Task Parallel Library in C#

Parallelism برای انجام چندین عملیات به صورت همزمان کاربرد دارد که سبب افزایش سرعت در انجام پردازش های برنامه ما می شود.

برنامه نویسی موازی به ما کمک می کند که task ها را به قسمت های مختلفی تقسیم کنیم و آن قسمت ها به صورت هم زمان با یک دیگر کارکنند.

مزیت اصلی : صرفه جویی در زمان با افزایش سرعت. با استفاده از هر تعداد thread که می توانیم برای انجام Task ها.

**نکته : عموما از parallelism برای cpu-bound ها استفاده می شود.**

**نکته:** باید توجه کرد که استفاده از parallelism همیشه نمی تواند مناسب باشد بلکه باید محاسبه شود که استفاده از آن واقعا دارای مزیت است یا نه؟ به دلیل این که دارای cost است که می تواند از مزایای آن بیشتر باشد.

در سیستم های که حجم داده های سنگینی را پردازش می کنند خیلی مهم است.

نکته مهم : باید حتما سخت افزار ما نیز قابلیت پشتیبانی از parallel را داشته باشد و گرنه به صورت seqencial عملیات انجام می شود.

امروزه بیشتر کامپیوتر ها از از پردازنده های چند هسته ای پشتیبانی می کنند که دارای هسته های logical نیز هستند که یعنی 4 هسته اصلی و 8 هسته منطقی که هر هسته 2 عملیات را می تواند انجام دهد.

**نکته » در پردازش موازی باید به تسک ها idempotent باشند و ترتیب اجرای مهم نباشد.**

به طور مثال می خواهیم برای یک کاربر sms و email ارسال کنیم می توانیم عملیات را به صورت parallel جلو ببریم.

# Parallel For in C#

نکته : در حلقه for به صورت parallel نمی توان انتظار خروجی به ترتیب صحیح مانند for معمولی داشت.

**static** **void** Main**(string[]** args**)**

**{**

Console.WriteLine**(**"C# Parallel For Loop"**)**;

//It will start from 1 until 10

//Here 1 is the start index which is Inclusive

//Here 11 us the end index which is Exclusive

//Here number is similar to i of our standard for loop

//The value will be store in the variable number

Parallel.For**(**1, 11, number =**>** **{**

Console.WriteLine**(**number**)**;

**})**;

Console.ReadLine**()**;

**}**

به دلیل این که فرآیند اجرا به صورت موازی در چندین thread انجام می شود هر بار نتیجه متقاوت است و به ترتیب نیست.

در حالتی که ترتیب مهم نیست و سرعت اجرا مهم است parallel مناسب است و همچنین با مثال زیر می توان ثابت کرد که parallel بر روی چند thread انجام می شود.

**static** **void** Main**(string[]** args**)**

**{**

Console.WriteLine**(**"C# For Loop"**)**;

**int** number = 10;

**for** **(int** count = 0; count **<** number; count++**)**

**{**

//Thread.CurrentThread.ManagedThreadId returns an integer that

//represents a unique identifier for the current managed thread.

Console.WriteLine**(**$"value of count = {count}, thread = {Thread.CurrentThread.ManagedThreadId}"**)**;

//Sleep the loop for 10 miliseconds

Thread.Sleep**(**10**)**;

**}**

Console.WriteLine**()**;

Console.WriteLine**(**"Parallel For Loop"**)**;

Parallel.For**(**0, number, count =**>**

**{**

Console.WriteLine**(**$"value of count = {count}, thread = {Thread.CurrentThread.ManagedThreadId}"**)**;

//Sleep the loop for 10 miliseconds

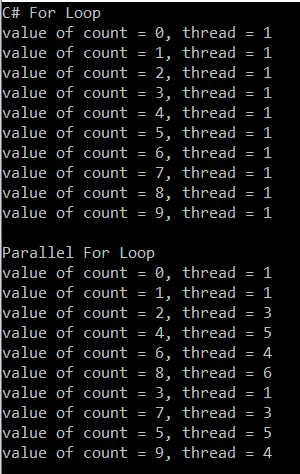
Thread.Sleep**(**10**)**;

**})**;

Console.ReadLine**()**;

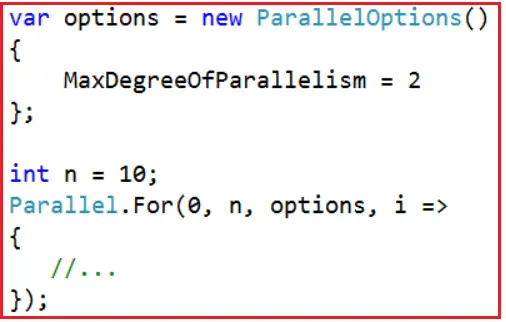
**}**

**}**



##### **ParallelOptions Class in C#**

به کمک این کلاس می توانیم مشخص کنیم که حداکثر چه تعداد thread می توانند در عملیات parallel شرکت کنند. به عبارتی چند عملیات هم زمان می تواند انجام شود.



##### **Terminating a Parallel For Loop in C#:**

Break <به معنی این که تکمیل کردن تمامی پیمایش ها در تمامی thread ها قبل از پیمایش جاری در thread جاری و سپس خروج از حلقه.

Stop به معنی متوقف کردن پیمایش در سریع ترین حالت ممکن.

Parallel.For**(**0, BreakSource.Count, **(**i, BreakLoopState**)** =**>**

**{**

BreakData += i;

**if** **(**BreakData **>** 100**)**

**{**

BreakLoopState.Break**()**;

Console.WriteLine**(**"Break called iteration {0}. data = {1} ", i, BreakData**)**;

**}**

**})**;

Parallel.For**(**0, StopSource.Count, **(**i, StopLoopState**)** =**>**

**{**

StopData += i;

**if** **(**StopData **>** 100**)**

**{**

StopLoopState.Stop**()**;

Console.WriteLine**(**"Stop called iteration {0}. data = {1} ", i, StopData**)**;

**}**

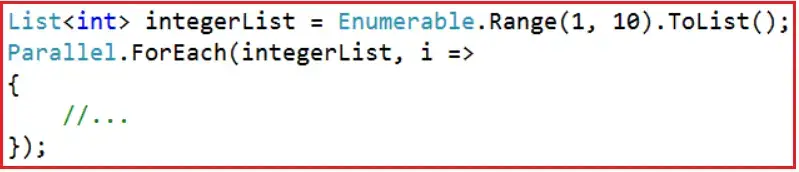
**})**;

**در parallel کلمات کلیدی breadk , exit کاربردی ندارد.**

* **Thread Safety:** Ensure that your loop code is thread-safe. Avoid modifying shared data without proper synchronization mechanisms like locks, concurrent collections, or other thread synchronization techniques.
* **Order of Execution:** The iterations of a Parallel For loop do not run in a guaranteed order, and different iterations may run simultaneously. This is different from a regular for loop where each iteration runs sequentially.
* **Exception Handling:** Handling exceptions in a Parallel For loop can be more complex. You might use a ParallelLoopState object to stop or break the loop, and an AggregateException may be thrown if multiple iterations throw exceptions.
* **Performance:** While Parallel.For can improve performance for CPU-bound operations, it might not be beneficial for I/O-bound operations. Additionally, for very short or simple loop bodies, the overhead of parallelization may outweigh its benefits.
* **Cancellation Support:** Parallel.For support task cancellation using the CancellationToken class. This can be useful for long-running operations that need to be stopped under certain conditions.
* **Return Values and Loop Control:** Parallel.For returns, a ParallelLoopResult which can be used to determine if the loop ran to completion or was stopped/prematurely exited. You can also control the loop execution using the ParallelLoopState object in the loop body.

# Parallel Foreach Loop in C#

همانند forech عادی عمل می کنند با این تقاوت که امکان پیمایش هم زمان و چندگانه بر روی چندین پردازش و هسته ی cpu را فراهم می کند.



این متد دارای ویژگی ها مشابه ی for برای محدود کردن تعداد thread ها است و همان مثال های for می تواند اینجا نیز کاربرد داشته باشد.

**نکته مهم در parallelism این است که حتما باید محاسبه کنید آیا بهینه است یا نه برای کار های بزرگ قطعا مناسب است اما برای کار های کوچک حالت sequencial مناسب تر است. به علت این که ایجاد هر thread خودش زمان بر است.**

**advantage**

**به طور مثال در large collection ها مناسب است. در cpu bound و idenpendent operations و performance سبب مقیاس پذیری می شود. منعطف و قابل تنظیم است. استفاده کارآمد از منابع دارد و ساده است.**

**Disadvantage**

Overhead در ایجاد thread و مدیریت و switching

Complex debug

Shared state handling سخت

Non deterministic behavior رفتار غیر قابل پیش بینی

عدم مزایا در i/o bound های

# Parallel Invoke in C#

یکی از متد های static پرکاربرد کلاس parallel است برای مواقعی که می خواهیم که متد های را به صورت موازی اجرا کنیم.

مثال : می خواهیم 3 متد را به صورت parallel اجرا کنیم.

**static** **void** Main**()**

**{**

Stopwatch stopWatch = new Stopwatch**()**;

stopWatch.Start**()**;

//Calling Three methods Parallely

Parallel.Invoke**(**

Method1, Method2, Method3

**)**;

stopWatch.Stop**()**;

Console.WriteLine**(**$"Parallel Execution Took {stopWatch.ElapsedMilliseconds} Milliseconds"**)**;

Console.ReadKey**()**;

**}**

**static** **void** Method1**()**

**{**

Thread.Sleep**(**200**)**;

Console.WriteLine**(**$"Method 1 Completed by Thread={Thread.CurrentThread.ManagedThreadId}"**)**;

**}**

**static** **void** Method2**()**

**{**

Thread.Sleep**(**200**)**;

Console.WriteLine**(**$"Method 2 Completed by Thread={Thread.CurrentThread.ManagedThreadId}"**)**;

**}**

**static** **void** Method3**()**

**{**

Thread.Sleep**(**200**)**;

Console.WriteLine**(**$"Method 3 Completed by Thread={Thread.CurrentThread.ManagedThreadId}"**)**;

**}**

**}**

اگر کد بالا را به صورت sequential اجرا کنیم مدت زمانی 2 برابر حالت parallel اجرای آن زمان می برد اما در این حالت زمان اجرا مناسب است اما باید در نظر گرفت که همیشه مناسب نیست مثال زمانی که کار کوچکی است صرفا هزینه بر است.

امکان محدود کردن تعداد thread های همانند حالت قبل نیز وجود دارد.

##### **Invoking Methods with Input and Return Type using Parallel.Invoke**

**static** **void** Main**()**

**{**

**int** intResult = 0;

**string** strResult = **string**.Empty;

//Calling Three methods Parallely

Parallel.Invoke**(**

**()** =**>** intResult = Method1**()**,

**()** =**>** strResult = Method2**(**"Pranaya"**)**,

**()** =**>** Method3**(**100**)**

**)**;

Console.WriteLine**(**$"Method1 Result: {intResult}"**)**;

Console.WriteLine**(**$"Method2 Result: {strResult}"**)**;

Console.WriteLine**(**$"Parallel Execution Completed"**)**;

Console.ReadKey**()**;

**}**

**static** **int** Method1**()**

**{**

Task.Delay**(**200**)**;

Console.WriteLine**(**$"Method 1 Completed by Thread={Thread.CurrentThread.ManagedThreadId}"**)**;

**return** 100;

**}**

**static** **string** Method2**(string** name**)**

**{**

Task.Delay**(**200**)**;

Console.WriteLine**(**$"Method 2 Completed by Thread={Thread.CurrentThread.ManagedThreadId}"**)**;

**return** "Hello:" + name;

**}**

**static** **void** Method3**(int** number**)**

**{**

Task.Delay**(**200**)**;

Console.WriteLine**(**$"Method 3 Completed by Thread={Thread.CurrentThread.ManagedThreadId}"**)**;

**}**

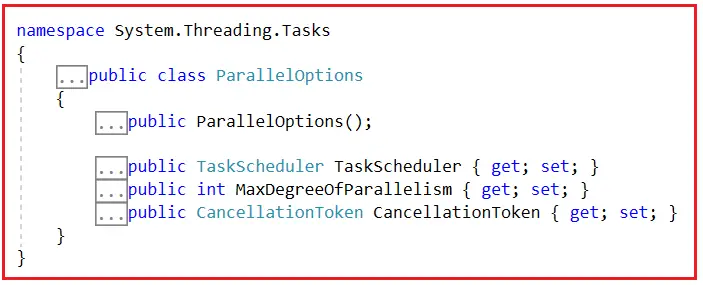
**}**

# Maximum Degree of Parallelism in C#

تا الان از تمامی منابع کامپیوتر خود برای انجام تسک ها استفاده می کردیم و به طوری که تا جایی که امکان داشت ترد می ساختیم در صورتی که در بیشتر مواقع به این تعداد thred نیاز نیست.

Parallel Option class

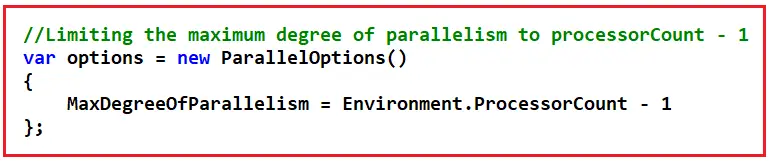
با استفاده از این کلاس می توان تعداد thread های که به صورت همزمان برای انجام یک Task به کار گرفته می شود را محدود کرد همچنین امکان cancell کردن یک Task را نیز فراهم کرده است.



1. **public TaskScheduler TaskScheduler {get; set;}:** It is used to get or set the TaskScheduler associated with this ParallelOptions instance. Setting this property to null indicates that the current scheduler should be used. It returns the task scheduler that is associated with this instance.
2. **public int MaxDegreeOfParallelism {get; set;}:** It is used to get or set the maximum number of concurrent tasks enabled by this ParallelOptions instance. It returns an integer that represents the maximum degree of parallelism. It will throw ArgumentOutOfRangeException if the property is being set to zero or to a value that is less than -1. -1 is the default value which sets that there is no limitation of the concurrent tasks to be executed.
3. **public CancellationToken CancellationToken {get; set;}:** It is used to get or set the CancellationToken associated with this ParallelOptions instance. It returns the token that is associated with this instance.

نکته » به صورت استاندارد باید تعداد حداکثر parallism را برابر با تعداد پردازنده ها منهای 1 کنیم. تعداد logical processor ها مهم است.

نکته : بهتر است به صورت hard code نباشد تعداد processor ها بلکه از کلاس c# آن استفاده شود.

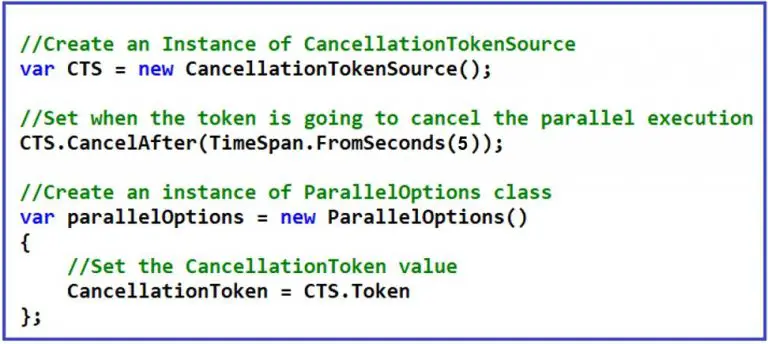


# How to Cancel Parallel Operations in C#

مانند برنامه نویسی async از Cancellation token برای ملقی کردن عملیات parallel استفاده کنیم

##### **How to Cancel Parallel Operations in C#?**

ابتدا لازم است که یک instance از کلاس CancellationTokenSource ایجاد کرده و سپس آن را به کلاس ParallelOption به عنوان property می دهیم.



**static** **void** Main**(string[]** args**)**

**{**

//Create an Instance of CancellationTokenSource

var CTS = new CancellationTokenSource**()**;

//Set when the token is going to cancel the parallel execution

CTS.CancelAfter**(**TimeSpan.FromSeconds**(**5**))**;

//Create an instance of ParallelOptions class

var parallelOptions = new ParallelOptions**()**

**{**

MaxDegreeOfParallelism = 2,

//Set the CancellationToken value

CancellationToken = CTS.Token

**}**;

**try**

**{**

Stopwatch stopwatch = new Stopwatch**()**;

stopwatch.Start**()**;

//Passing ParallelOptions as the first parameter

Parallel.Invoke**(**

parallelOptions,

**()** =**>** DoSomeTask**(**1**)**,

**()** =**>** DoSomeTask**(**2**)**,

**()** =**>** DoSomeTask**(**3**)**,

**()** =**>** DoSomeTask**(**4**)**,

**()** =**>** DoSomeTask**(**5**)**,

**()** =**>** DoSomeTask**(**6**)**,

**()** =**>** DoSomeTask**(**7**)**

**)**;

stopwatch.Stop**()**;

Console.WriteLine**(**$"Time Taken to Execute all the Methods : {stopwatch.ElapsedMilliseconds/1000.0} Seconds"**)**;

**}**

//When the token cancelled, it will throw an exception

**catch(**Exception ex**)**

**{**

Console.WriteLine**(**ex.Message**)**;

**}**

**finally**

**{**

//Finally dispose the CancellationTokenSource and set its value to null

CTS.Dispose**()**;

CTS = **null**;

**}**

Console.ReadLine**()**;

**}**

# Atomic Methods Thread Safety and Race Conditions in C#

##### **Atomic Methods in C#:**

معمولا به این صورت است که هر thread به صورت idenpendent کار خود را انجام می دهد اما بعضی مواقع نیاز است که اطلاعات بین thread ها تقسیم شود.

یک روش مهم atomic method است که این تضمین را می دهد که ما همیشه نتیجه یکسانی را بدست می آوریم. مهم نیست که چند thread به صورت هم زمان method را اجرا کنند.

2 ویژگی مهم

* هیچ thread نمی تواند حالت میانی متدی را ببیند که عملیات شروع نشده یا قبلا کامل شده است هیج حالت میانی بین شروع و پایان نیست
* یا عملیات با موفقیت به پایان می رسد یا شکست می خورد مانند transaction در پایگاه داده.

##### **How to Achieve Atomicity in C#?**

راه های زیادی وجود دارد یکی از ان ها استفاده از lock است که این امکان را فراهم می سازد که سایر thead ها به قسمت کد ما زمانی که lock فعال است دسترسی نداشته باشند یا در مواقع استفاده از collection ها از نوع concurrent استفاده کنیم.

##### **Thread Safety in C#**

زمانی میگوییم که یک متد thread safe است که می توانیم متد را به صورت هم زمان توسط تردهای اجرا کنیم بدون error زمانی به این مفهوم می رسیم که دیتای ما به مشکل نخورد و بیش از یک thread بر روی یک داده کاری نکنند.

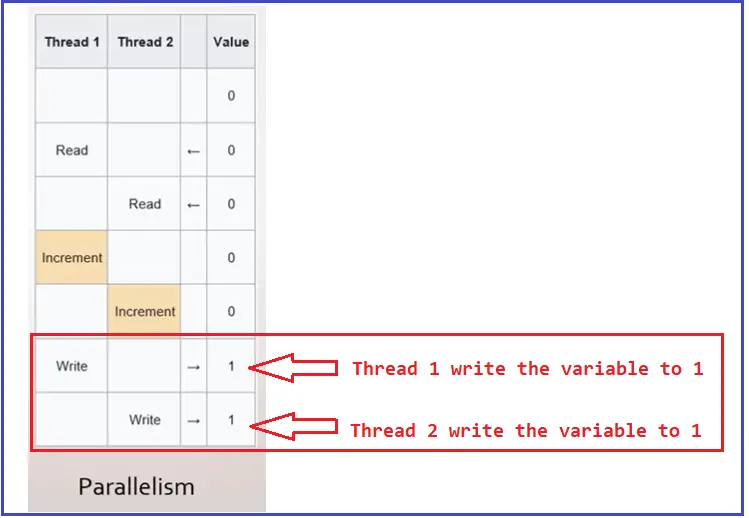
##### **How to Achieve Thread Safety in C#?**

با استفاده از lock و امکانات مشابه می توان به thread safety رسید. اگر نیاز به انتقال object داشته باشیم می توانیم از immutable ها استفاده کنیم.

##### **Race Conditions in C#:**

زمانی رخ می دهد که یک متغییر توسط چندین Thread بخواهد تغییر کند به صورت هم زمان مشکل زمانی رخ می دهد که ترتیب انجام عملیا ت برای ما مهم باشد. مثلا زمانی که بخواهیم به یک متغییر 1 اضافی کنیم

مثال این اتفاق در تصویر زیر آمده است که 2 ترد به صورت همزمان می خواهند مقدار یک variable را تغییر دهند.



##### **How to Solve the above Problem in C#?**

با استفاده از کلاس interlock می توانیم مشکل فوق را حل کنیم.

##### **Interlocked in C#:**

این کلاس به ما این اجازه را می دهد که یک عملیات را به صورت atomic انجام دهیم به این صورت که عملیات را Safe با استفاده از چندین thread انجام دهیم. یعنی امکاناتی مدهد که ما را مجاز می کند که عملیاتی که توسط چندین thread هم زمان انجام می شود را به صورت safe انجام دهیم.

مثال:

انتظار داریم متغییر زیر به عدد 10000 برسد با هر بار اجرای حلقه که 10000 بار تکرار و افزایش 1 مقدار را انجام میدهد.

var ValueWithoutInterlocked = 0;

Parallel.For**(**0, 100000, \_ =**>**

**{**

//Incrementing the value

ValueWithoutInterlocked++;

**})**;

خروجی با 4 بار اجرا > 27412 و 47071 و 61572 و 74321

حال بازنویسی کد با کلاس interlock که کلاسی استاتیک است.

var ValueInterlocked = 0;

Parallel.For**(**0, 100000, \_ =**>**

**{**

//Incrementing the value

Interlocked.Increment**(ref** ValueInterlocked**)**;

**})**;

به نتیجه دلخواه می رسیم. این کلاس متد های فراوانی دارد که می توان از آن ها استفاده کرد.

##### **Lock in C#:**

نکته مهمی که دارد این است که thread های که wating هستند را lock می کند بنا براین باید توجه داشت که عملیات خیلی طولانی نباشد که سبب کاهش سرعت به دلیل lock کردن thread ها می شود.

**static** **object** lockObject = new **object()**;

var ValueWithLock = 0;

Parallel.For**(**0, 100000, \_ =**>**

**{**

**lock(**lockObject**)**

**{**

//Incrementing the value

ValueWithLock++;

**}**

**})**;

# Interlocked vs Lock in C#

مشکل ما زمانی بود که یک متغییر بین چندین thread به صورت مشترک استفاده می شده و به دلیل ترتیب عملیات متفاوت بین thread ها هر مقدار variable ما با مقدار مورد انتظار متفاوت می شده.

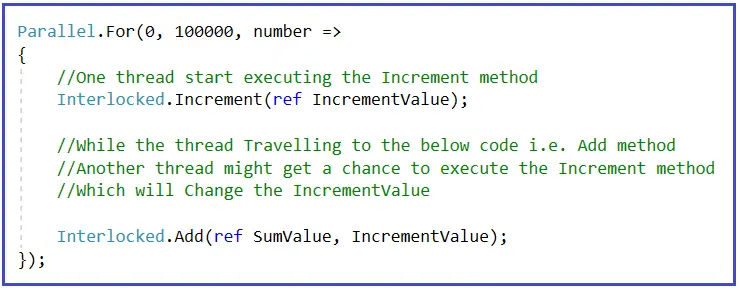
برای افزایش 1 مقدار به متغییر عملیات های read ,increasing , write داریم که 2 ترد هم زمان انجام می دهند.

Methods

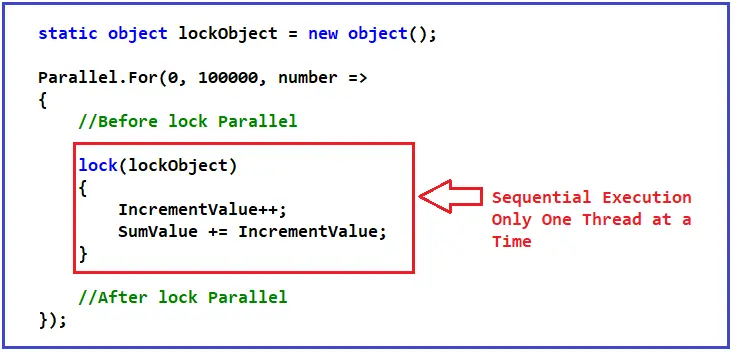
1. **Increment():** This method is used to increment a variable’s value and store its result. Int32 and Int64 integers are its legal parameters.
2. **Decrement():** This method is used to decrement a variable’s value and store its result. Int32 and Int64 integers are its legal parameters.
3. **Exchange():** This method is used to exchange values between variables. This method has seven overloaded versions based on the different types it can accept as its parameter.
4. **CompareExchange():** This method compares two variables and stores the result of the comparison in another variable. This method also has seven overloaded versions.
5. **Add():** This method is used to add two integer variables and update the result in the first integer variable. It is used to add integers of type Int32 as well as Int64.
6. **Read():** This method is used to reads an integer variable. It is used to read an integer of type Int64.
7. **public static long Add(ref long location1, long value):** This method adds two 64-bit integers and replaces the first integer with the sum, as an atomic operation.
8. **public static int Add(ref int location1, int value):** This method adds two 32-bit integers and replaces the first integer with the sum, as an atomic operation. It returns the new value stored at location1.

در حالت کلی interlock از Lock سریع تر است. در شرایط مشابه پیشنهاد می شود از interlock استفاده شود مگر این که عملیاتی باشد که از پس آن برنیاید.

نکته 2 عملیات هم زمان با interlock سبب بروز نتیجه غیرمنتظره می شود به دلیل این که 2 ترد مستفل هر قسمت رو دارند اجرا می کنند در این حالت lock موثر است. استفاده از مکانیسم asynchronous



Solve



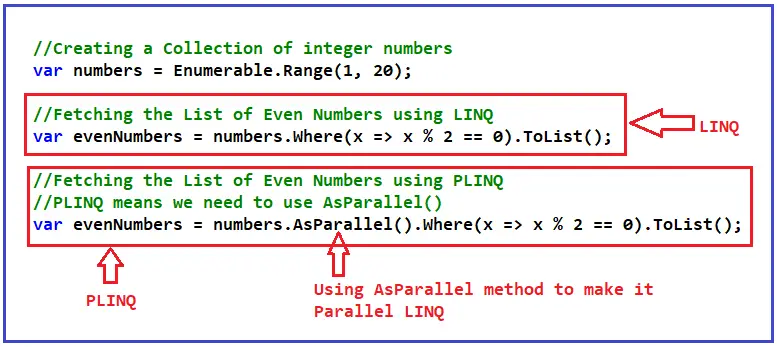
# Parallel LINQ in C#

اگر collection از داده ها داشته باشیم و بخواهیم بر روی آن پردازشی انجام دهیم می توانیم از plinq استفاده کنیم که کاملا مشابه linq است با این تفاوت که می تواند parallel است.

قابلیت استفاده از حداکثر درجه ی parallelism و cancellationToken را نیز دارا است. برای استفاده از مزایای آن می تواند از متد AsParallel استفاده کرد.

کاربرد : مواقعی که حجم داده زیاد است و نوع عملیات ما cpu-bound است کاربرد دارد.

کالکشن را به سگمنت های تقسیم می کند و داده های هر سگمنت توسط thread ها پردازیش می شود.



نکته مهم : در حالت sequential ترتیب درستی به ما مددهد اما در حالت parallel چون هر Thread به صورت idempotent دارد کار می کند ترتیب بهم می خورد در خروجی.

##### **How to Maintain the Original Order in Parallel LINQ?**

با استفاده از متد AsOrdered()

//Creating a Collection of integer numbers

var numbers = Enumerable.Range**(**1, 20**)**;

//Fetching the List of Even Numbers using PLINQ

//PLINQ means we need to use AsParallel()

var evenNumbers = numbers

.AsParallel**()** //Parallel Processing

.AsOrdered**()** //Original Order of the numbers

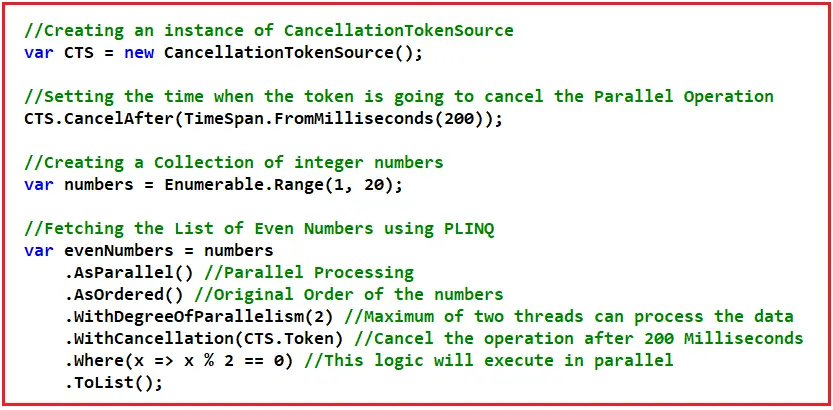
.Where**(**x =**>** x % 2 == 0**)**

.ToList**()**;

##### **Difference Between OrderBy and AsOrdered Method in C#:**

تفاوت در این است که orderBy داده ها مرتب می کند اما AsOrdered داده ها را مرتب نمی کند بلکه ترتیب اصلی و اولیه را حفظ می کند

##### **Maximum Degree of Parallelism and Cancellation Token in Parallel LINQ**



##### **Doing Aggregates in Parallel LINQ**

var numbers = Enumerable.Range**(**1, 10000**)**;

//Sum, Min, Max and Average LINQ extension methods

Console.WriteLine**(**"Sum, Min, Max and Average with LINQ"**)**;

var Sum = numbers.AsParallel**()**.Sum**()**;

var Min = numbers.AsParallel**()**.Min**()**;

var Max = numbers.AsParallel**()**.Max**()**;

var Average = numbers.AsParallel**()**.Average**()**;

Console.WriteLine**(**$"Sum:{Sum}\nMin: {Min}\nMax: {Max}\nAverage:{Average}"**)**;

Console.ReadKey**()**;

performance

**static** **void** Main**()**

**{**

var random = new Random**()**;

**int[]** values = Enumerable.Range**(**1, 99999999**)**

.Select**(**x =**>** random.Next**(**1, 1000**))**

.ToArray**()**;

//Min, Max and Average LINQ extension methods

Console.WriteLine**(**"Min, Max and Average with LINQ"**)**;

Stopwatch stopwatch = new Stopwatch**()**;

stopwatch.Start**()**;

// var linqStart = DateTime.Now;

var linqMin = values.Min**()**;

var linqMax = values.Max**()**;

var linqAverage = values.Average**()**;

stopwatch.Stop**()**;

var linqTimeMS = stopwatch.ElapsedMilliseconds;

DisplayResults**(**linqMin, linqMax, linqAverage, linqTimeMS**)**;

//Min, Max and Average PLINQ extension methods

Console.WriteLine**(**"\nMin, Max and Average with PLINQ"**)**;

stopwatch.Restart**()**;

var plinqMin = values.AsParallel**()**.Min**()**;

var plinqMax = values.AsParallel**()**.Max**()**;

var plinqAverage = values.AsParallel**()**.Average**()**;

stopwatch.Stop**()**;

var plinqTimeMS = stopwatch.ElapsedMilliseconds;

DisplayResults**(**plinqMin, plinqMax, plinqAverage, plinqTimeMS**)**;

Console.ReadKey**()**;

**}**

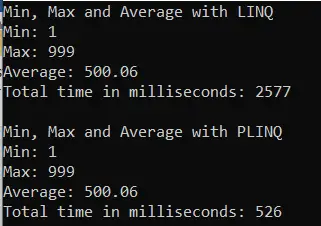
**static** **void** DisplayResults**(int** min, **int** max, **double** average, **double** time**)**

**{**

Console.WriteLine**(**$"Min: {min}\nMax: {max}\n" + $"Average: {average:F}\nTotal time in milliseconds: {time}"**)**;

**}**

**}**



* **Ordering:** PLINQ processes elements in parallel and doesn’t guarantee the order of the results unless you explicitly ask for it using the AsOrdered method. Preserving order can add overhead and potentially reduce performance gains.
* **Side Effects:** Queries should be side-effect-free. Since PLINQ can process multiple elements simultaneously, using functions that have side effects can result in unexpected behavior or race conditions.
* **Exception Handling:** When an exception is thrown in a PLINQ query, it is wrapped in an AggregateException. This exception contains all the individual exceptions thrown during the execution of the query.
* **Performance:** Not all queries run faster in parallel. PLINQ has overhead, so for small collections or fast operations, the overhead might outweigh the benefits of parallelism. Measuring performance for sequential and parallel versions of your query is generally a good idea.
* **Forcing Parallelism:** You can force PLINQ to parallelize the query using the WithExecutionMode method with ParallelExecutionMode.ForceParallelism is not recommended unless you are certain that parallelism will improve performance.
* **Degree of Parallelism:** You can specify the number of processors to use in a PLINQ query by using the WithDegreeOfParallelism method. This can be useful for fine-tuning performance, particularly if you want to leave some cores free for other tasks.

##### **When to Use Parallel LINQ in C#?**

* **Large Data Sets:** When processing large data collections, PLINQ can significantly reduce the time it takes by dividing the work across multiple threads.
* **CPU-Intensive Operations:** If your LINQ queries involve complex calculations or CPU-bound processing, PLINQ can help distribute the load across multiple cores.
* **Multi-Core Processors:** If the environment where your application is running has multiple CPU cores available, PLINQ can utilize these cores to perform operations in parallel.
* **Long-Running Queries:** For operations that naturally take a long time to execute, parallelizing the work can lead to better application responsiveness, especially in desktop or server environments.

##### **When Not to Use Parallel LINQ in C#?**

* **Small Data Sets:** If the collection is relatively small, the overhead of partitioning the data and managing threads may outweigh the benefits of parallel processing.
* **I/O-Bound Operations:** If your query is I/O-bound, waiting on file system operations or network responses, parallelizing the query won’t improve performance and may even degrade it.
* **Ordered Queries:** When the order of results is important, PLINQ can introduce overhead because maintaining order in a parallelized environment is more complex.
* **Non-Thread-Safe Operations:** If your query performs operations that are not thread-safe, such as writing to shared variables without proper synchronization, PLINQ can cause more problems than it solves.
* **Asynchronous Operations:** If you can perform I/O-bound work asynchronously, doing so is often more efficient with async/await rather than using PLINQ.

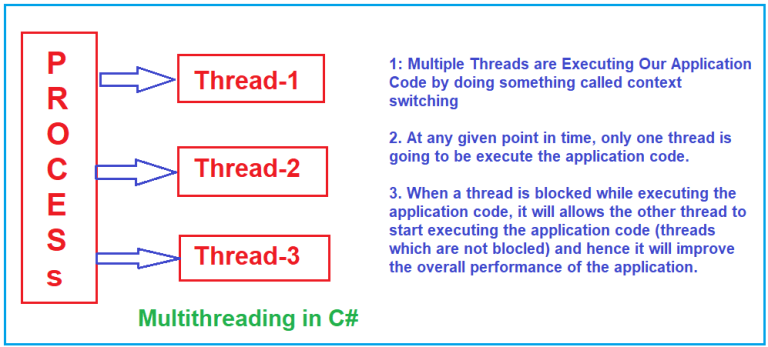
# Multithreading vs Asynchronous Programming vs Parallel

1. **Multithreading:** This is all about a single process split into multiple threads.
2. **Parallel Programming:** This is all about multiple tasks running on multiple cores simultaneously.
3. **Asynchronous Programming:** This is all about a single thread initiating multiple tasks without waiting for each to complete.

##### **What is Multithreading in C#?**

قابلیت ایجاد و مدیریت چندین thread در یک پردازش است. یک thread کوچکترین واحد پردازشی است و چندین thread می توان به صورت همزمان اجرا شوند و تقسیم کنند منابع پردازش پدر خود را اما هر کدام کد متفاوتی را اجرا کنند.

تمام اپلیکیشن های سی شارپی با یک نخ به نام main اجرا می شوند.



threadPool ارائه دهنده استخری از نخ های آماده به کار است که می توانند برای انجام کار ها استفاده شوند.

Advantage

در تسک های لانگ رانینگ می تواند سبب عدم بلاک شدن ui شود.

بهره وری در پردازنده های چنده هسته بیشتر و بهتر است.

**Challenges:**

**RaceCondition زمانی که 2 ترد به یک داده مشترک برسند و بخواهند هم زمان تغییر دهند.**

**DeadLock : زمانی که 2 یا چند نخ منتظر آزاد شدن منابع می مانند.**

**Resource starvation : زمانی که یک نخ دسترسی به منبع ندارد و نمی تواند کارش را انجام دهد.**

**راه حل ها**

Mutex, Monitor, Semaphore, and lock

**Considerations**

ساخت تعداد زیادی نخ می تواند عملکرد را کاهش دهد به لیل context switching overhead

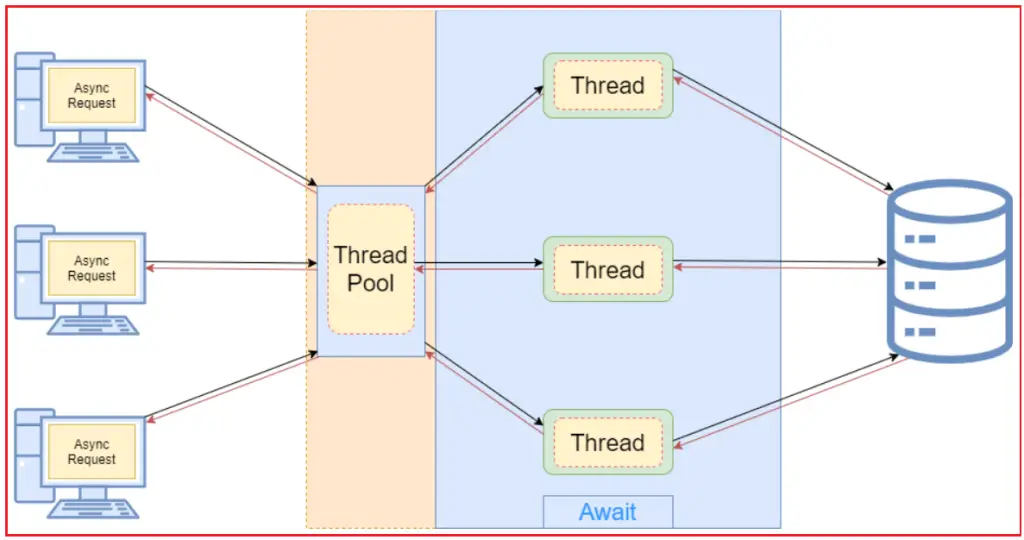
نخ ها منابع را درگیر می کنند و سبب کاهش سرعت می تواند بشد

هم زمانی می تواند سربار داشته باشد بنابراین لازم است که تعادل رعایت شود.

چند نخی یک ویژگی قدرتمند است اما باید در طراحی و ساتفاده و تست هوشیار بود.

##### **What is Asynchronous Programming in C#?**

یک روش برنامه نویسی است که در آن یک تسک بدون بلاک کردن سایر ترد ها و ترد اصلی کار می کند که بخصوص در io ها بسیار کاربردی است به دلیل این که انتظار برای پایان کار می تواند زمان پردازند را هدر بدهد بلکه بهتر است کار های دگیر انجام شود تا عملیات تکمیل می شود.



زمانی که درخواست به اپلیکیشن می آیند سرور از thread pool یک ترد برمی دارد و کد را اجرا می کند اما نکته مهم این است که ترد عملیات io را بخواهد انجام دهد ترد منتظر تکمیل شدن نمی ماند یعنی بلاک نمی شود بلکه به ترد پول برمیگردد و می تواند سایر درخواست ها را مدیریت کند که سبب افزایش بهره وری پردانزده می شودکه سبب پاسخ دهی بیشتر به درخواست های کلاینت می شود.

base

کلمه کلیدی Async , await 2 کلمه کلیدی مهم در پیاده سازی async است که برای متد های که task , task<t> برمیگردانند کاربرد دارد کلمه Await به کامپایلر می گوید که اگر task تکمیل نشده است بگذار سایر کار ها انجام شوند.

**Core Classes and Namespaces:**

**Task برای عملیات های async که می توانند await کنند کاربرد دارد**

**Task<T> برای عملیات های که قابلیت await دارند و مقداری T را برمیگردانند**

**TaskCompletionSource<T> امکان ایجاد عملیات async که بر اساس async , await نیست**

**Advantages**

**پاسخ گو نگه داشتن ui**

**افزایش مقیاس پذیری اپلیکیشن و امکان پاسخ دهی تعداد درخواست بیشتر با آزاد کردن thread ها.**

**Considerations**

**این نوع برنامه نویسی به معنای multithreading نیست بلکه به معنای افزایش بهره وری در استفاده از thread ها است.**

**از async void برهیز شود به لیل این که نمی تواند await شود و همچنین بروز exception نمی تواند مدیریت شود**

**دیباگ کردن این نوع کد نویسی می تواند مقداری چالش برانگیز باشد.**

##### **What is Parallel Programming in C#?**

Process یا فرآیند استفاده از concurrency برای اجرای چندین محاسبه به صورت هم زمان برای بهبود عملکر و پاسخگویی نرم افزار است.

یک task را به چندین process تقسیم می کنیم و هر کدام را توسط core های cpu اجرا می کنیم به کمک thread.

**Basics:**

**Data parallelism: انجام عملیات parallel در collection های داده**

**Task parallelism : اجرای چندین Task به صورت موازی parallel تسک ها می توانند متمایز یا distinct باشند.**

**Advantages:**

**تسک های محاسباتی با مشارکت چندین processor یا Core می توانند سریع تر تکمیل شوند : Performace**

**افزایش پاسخگویی نرم افزار در تسک های log running**

**Considerations**

**دارای overhead است لازم نیست تمامی task ها به صورت parallel اجرا شوند.**

**Ordering ترتیب در پردازش parallel نباید مهم باشد اگر مهم است باید عملیات اضافی انجام شود که دارای هزینه است**

**Synchronization : در حالت های که shared data access داریم مکانیسم های synchronization مثل lock کار آمد هستند یا استفاده از concurrent collection ها.**

**Max degree of parallelism امکان محدود کردن سطح تعداد task های مورد استفاده به صورت هم زمان.**

**Differences from Asynchronous Programming:**

**برنامه نویسی parallel , synchronous در رابطه با بحث concurrency توافق دارند اما هدف نهایی متفاوت است.**

**Parallel از چندین core استفاده می کند برای افزایش عملکرد در کار های computional در حالی که asynchronous در رابطه را ارتقای responsiveness یا پاسخ گویی است و جلوگیری از wating یا صبر برای تکمیل عملیات های که به صورت long running هستند.**

**نکته مهم**

**پیاده سازی هر گونه عملیات parallel نیازمند دانش و بررسی زیاد می باشد برای این که آیا واقعا با حالت sequential دارای تفاوت و بهبود عملکرد است یا نه ؟**

#### **Multithreading vs. Asynchronous Programming vs. Parallel Programming in C#**

**MultiThreading**

تعریف : چند نخی به معنای اجازه دادن به یک process جهت مدیریت چنیدن thread است. که کوچکترین واحد های اجرایی cpu هستند و هر thread می تواند مجموعه ای از دستورات را اجرا کند

**برای عملیات های cpu , io می تواند مناسب باشد برای تسک های که می توانند به صورت concurrent یا هم زمان اجرا شوند و نیازی به صبر برای تکمیل ندارند و تسک های که می توانند به قسمت های کوچکتری تقسیم شوند قسمت های مستقل.**

مزایا : اگر به درستی استفاده شود در cpu های multicore کارآمد است.

معایب : thread ها مصرف کننده منابع هستند مدیریت دستی آن ها می تواند سخت باشد سبب deadlock , race condition شود و افزایش context switching نیز بر روی عملکرد تاثیر منفی دارد.

##### **Asynchronous**

تعریف: به سیستم ما این اجازه را می دهد که بر روی یک task کار کند بدون منتظر بودن برای تکمیل تسک دیگر بیشتر در رابطه با flow control یا کنترل جریان در اجرا های concurrent است. این نوع برنامه نویسی در C# بسیاز straightforward است.

استفاده : به صورت مخصوص برای i/o ها مثل دیتا بیس وب سرویس و فایل زمانی که نمی خواهیم wait یا block اتفاق بیوفتد عمیات های که می خواهیم قبل از انجام تسک های دیگر انجام شوند زمانی که میخواهیم در انتظار جواب هستیم عملیات های دیگر انجام شوند.

مزایا : پاسخ گویی رابط کاربری و مدیریت اسان نسبت به thread .

معایب : دیباگ نسبتا سخت و نیاز به دانش و پیچیدگی نسبی.

##### **Parallel**

تعریف : به معنای تقسیم یک task به تعدادی sub task به پردازش آن ها به صورت هم زمان انجام بپذیرد معمولا به کمک چند پردازنده یا core که تسک بین آن ها distribute می شود.

استفاده : عمیات های cpu که می توانند به زیر تسک های مستقل تقسیم شوند و به صورت هم زمان پردازش شوند زمانی که حجم داده برای پردازش زیاد است.

مزایا : افزایش سرعت با استفاده از ظرفیت هسته های در دسترس cpu

معایب : دارای overhead است و مناسب همه ی task ها نیست در share resource , race condition نیاز به بررسی دارد.

نکته : برنامه نویسی parallel و multithread معمولا دست در دست هم می دهند زیرای کای های parallel در thread های separate یا جداگانه اجرا می شوند.

نکته : برنامه نویسی Async می تواند multithread باشد به خصوص زمانی که یک تسک به thread جداگانه بارگذاری می شود.

**Choosing the Right Approach:**

* **IO-Bound Operations:** Asynchronous programming is best suited for IO-Bound Operations.
* **CPU-Bound Operations:** Parallel Programming and Multithreading best suitable for CPU-Bound Operations.

همیشه overhead مربوط به context switching در نظر گرفته شود در حالت parallel , multithread.

* **Multithreading** focuses on allowing multiple threads to operate concurrently, often within a single process, to maximize resource usage and maintain responsiveness. So, Multithreading is a process that contains multiple threads within a single process. Here, each thread performs different activities.
* **Asynchronous Programming** focuses on non-blocking operations, especially for I/O-bound tasks, ensuring responsiveness and scalability.
* **Parallel Programming** focuses on splitting tasks to run simultaneously on multiple processors or cores to reduce total computation time for CPU-bound operations. In this case, it will use multiple processors to execute different parts of a task; each processor has multiple threads that can execute the application code.