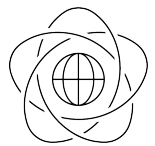
Programming 3 Advanced

Parallel, Concurrent Collections, Concurrency Workshop

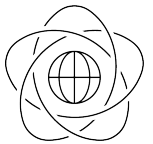
Tomasz Herman

Faculty of Mathematics and Information Science
Warsaw University of Technology

Lecture 13, 12 stycznia 2025



Outline



Parallel Class

Parallel.Invoke Invokes an array of delegates in parallel

Parallel.For Parallel version of for loop

Parallel.ForEach Parallel version of foreach loop

- The work is partitioned efficiently into a few tasks
- All methods block until all work is complete
- In case of exception workers are stopped and exception is rethrown and wrapped in an AggregateException

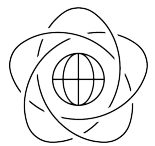
Parallel class

`https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallel?view=net-9.0`

Parallel.Invoke

```
1 Parallel.Invoke(
2 () => DownloadFile("https://foo.com", "foo.html"),
3 () => DownloadFile("https://bar.net", "bar.html"),
4 () => DownloadFile("https://microsoft.com", "ms.html"));
```

- It will work efficiently even with larger number of delegates, as the delegates are partitioned
- When using Parallel.*, you must ensure thread-safe access to shared variables

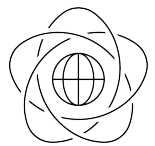


Parallel.For

Simplest signature:

```
1 public static ParallelLoopResult For(
2     int fromInclusive, int toExclusive, Action<int> body)
```

```
1 Parallel.For(from, to, i =>
2 {
3     Console.WriteLine($"Is {i} prime: {IsPrime(i)}");
4 });
```

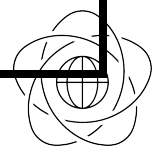


Parallel.ForEach

Simplest signature:

```
public static ParallelLoopResult ForEach<TSource>(
    IEnumerable<TSource> source, Action<TSource> body)
```

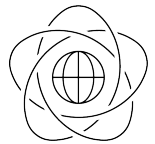
```
List<(string url, string output)> res =
[
    // ...
];

Parallel.ForEach(res, ((string url, string output) tuple) =>
{
    DownloadFile(tuple.url, tuple.output);
});
```



Breaking early

```
1  var res = Parallel.For(10000, 10010, (i, loopState) =>
2  {
3      if (IsPrime(i))
4      {
5          loopState.Break();
6      }
7  }); // result is ParallelLoopResult
8  if (!res.IsCompleted)
9      Console.WriteLine($"Prime: {res.LowestBreakIteration}");
10 else
11     Console.WriteLine("There are no primes in this range");
```

- Both For/ForEach have overloads that expose ParallelLoopState.
- Break: Ensures at least the same iterations as sequential execution.
- Stop: Immediately halts threads after the current iteration.
- Examining ParallelLoopResult object tells whether the loop ran to completion



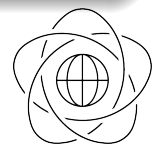
Concurrent Collections

- ConcurrentStack<T>
- ConcurrentQueue<T>
- ConcurrentBag<T>
- ConcurrentDictionary<TKey,TValue>
- BlockingCollection<T>

There is no concurrent version of the List.

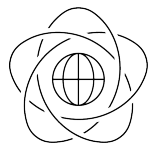
Thread-safe collections

`https://learn.microsoft.com/en-us/dotnet/standard/collections/thread-safe/`



Concurrent Collections Properties

- Conventional collections outperform concurrent collections except in highly concurrent scenarios.
- Enumerating over a concurrent collection during modifications by another thread mixes old and new content, without throwing exceptions.
- Concurrent stack, queue, and bag classes use linked lists internally.

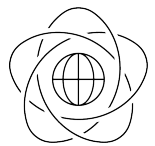


IProducerConsumerCollection

```
1  public interface IProducerConsumerCollection<T> :
2      IEnumerable<T>, ICollection
3  {
4      bool TryAdd(T item);
5      bool TryTake(out T item);
6      T[] ToArray();
7  }
```

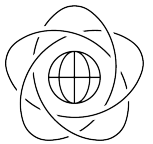
Classes that implement the interface:

- ConcurrentStack<T>
- ConcurrentQueue<T>
- ConcurrentBag<T>



ConcurrentBag

- ConcurrentBag is an unordered collection of objects
- Inside a concurrent bag, each thread gets its own private linked list
- Calling Take on a concurrent bag is very efficient — there are no races as long as each thread doesn't take more elements than it Added



BlockingCollection

- A blocking collection wraps any collection that implements `IProducerConsumerCollection<T>` and lets you Take an element from the wrapped collection — blocking if no element is available.
- A blocking collection also lets you limit the total size of the collection, blocking the producer if that size is exceeded.
- If constructor is called without passing in a collection, the class will automatically use a `ConcurrentQueue<T>`
- Another way to consume elements is to call `GetConsumingEnumerable`. This returns a (potentially) infinite sequence that yields elements as they become available. You can force the sequence to end by calling `CompleteAdding`: this method also prevents further elements from being enqueued.