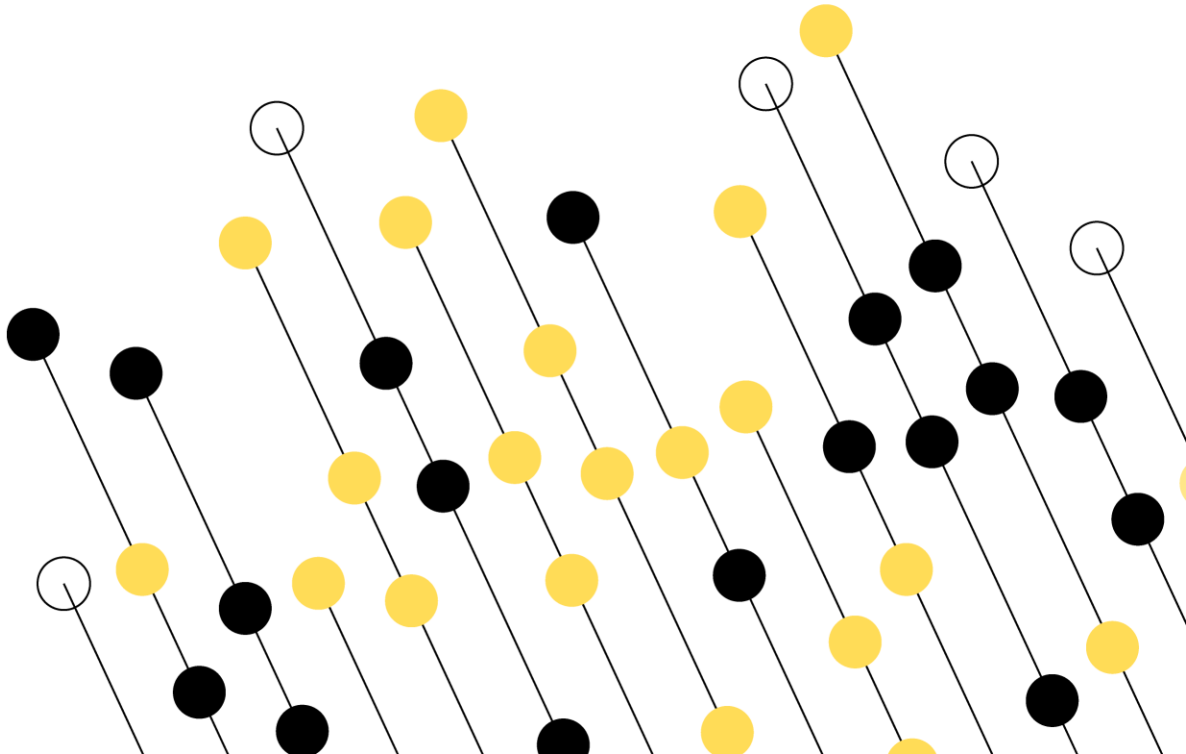
C# threads

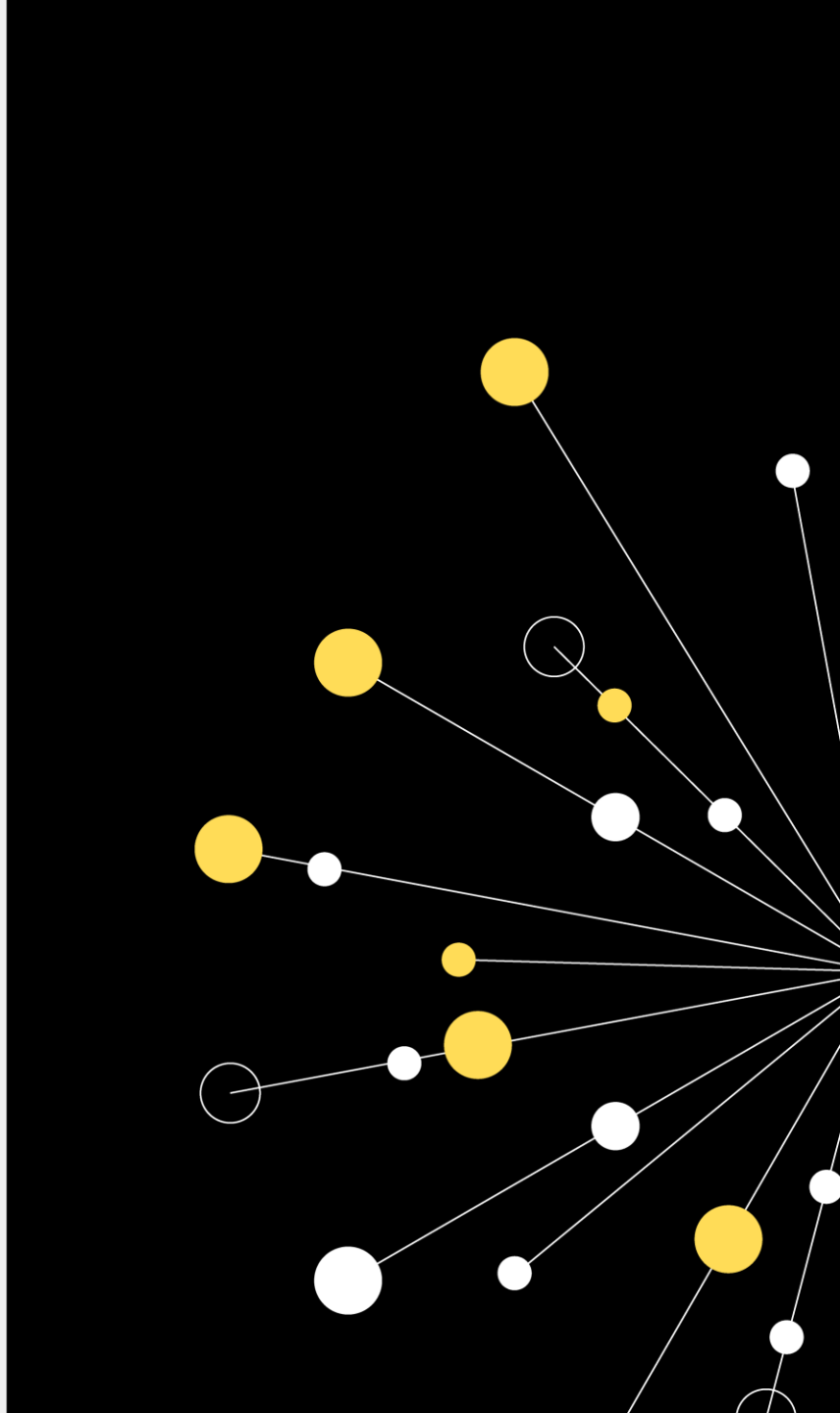


Running a Simple Task

Task and asynchronous execution.

```
Task task = Task.Run(() => {
    Console.WriteLine("Task started...");
    Task.Delay(1000).Wait();
    Console.WriteLine("Task finished!");
});
task.Wait();
```

Task.Run()
Starts background work.
Main thread continues while the task runs.



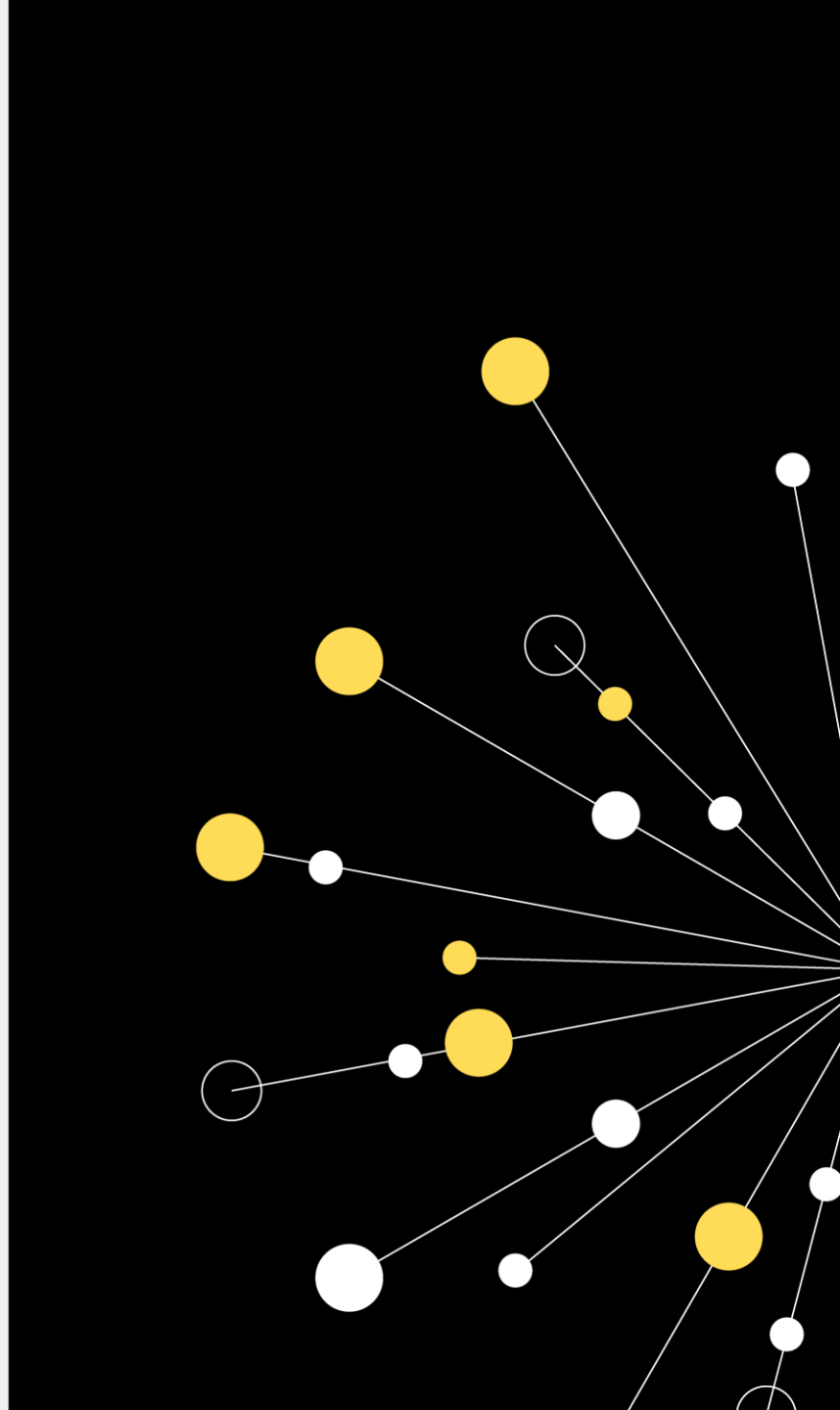
QA

Returning a Value from a Task

Run a background calculation that returns a result.

```
Task<int> calcTask = Task.Run(() => {
    int sum = 0;
    for (int i = 1; i <= 5; i++)
        sum += i;
    return sum;
});
int result = calcTask.Result;
```

Task<T>
can return data. **Accessing .Result**
waits for completion.



QA

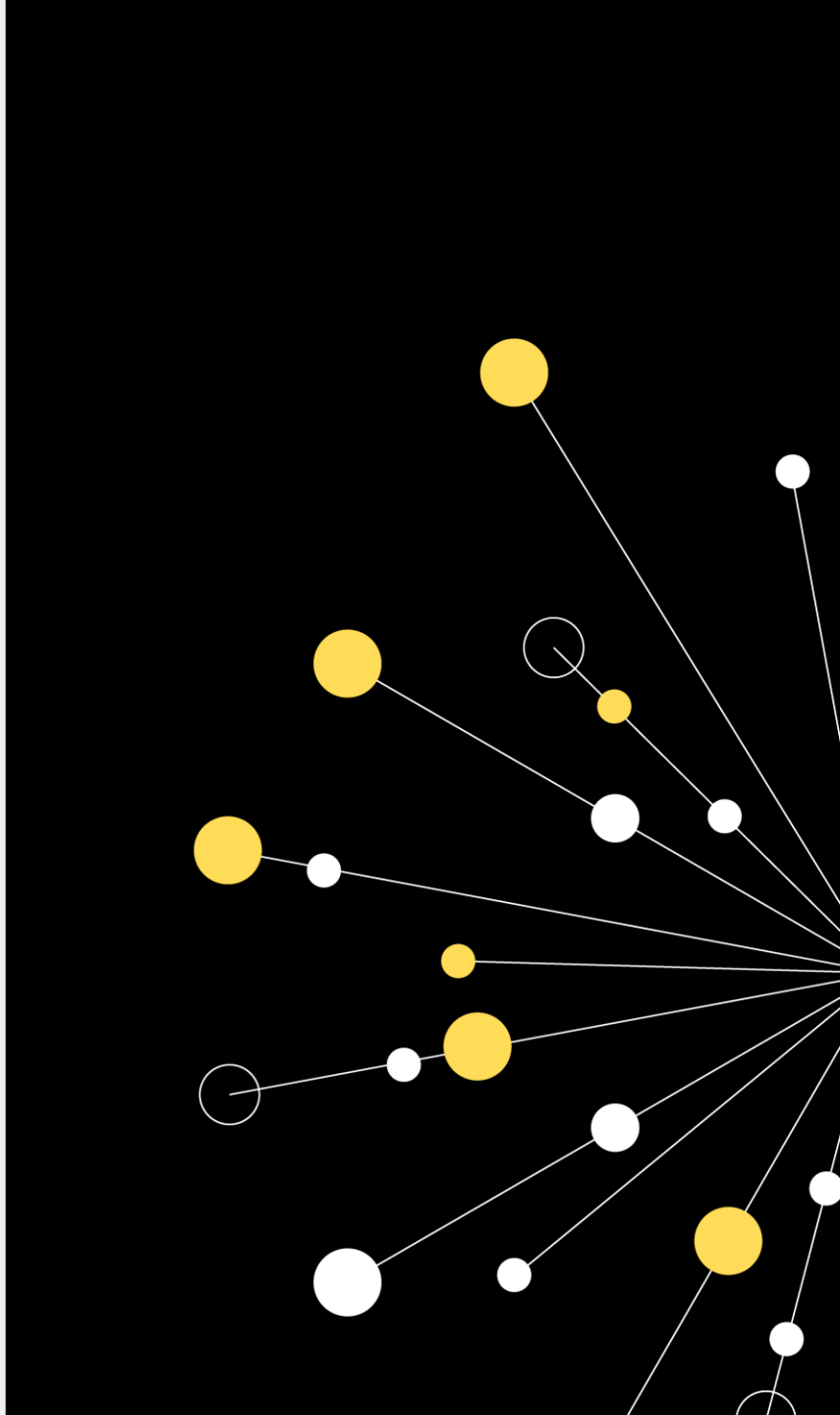
Multiple Tasks and WhenAll

Goal: Run several tasks in parallel and await all results.

```
Task<int> t1 = Task.Run(() => 10);
Task<int> t2 = Task.Run(() => 20);

int[] results = await Task.WhenAll(t1, t2);
```

Task.WhenAll
waits for multiple tasks.
Ideal for parallel operations.



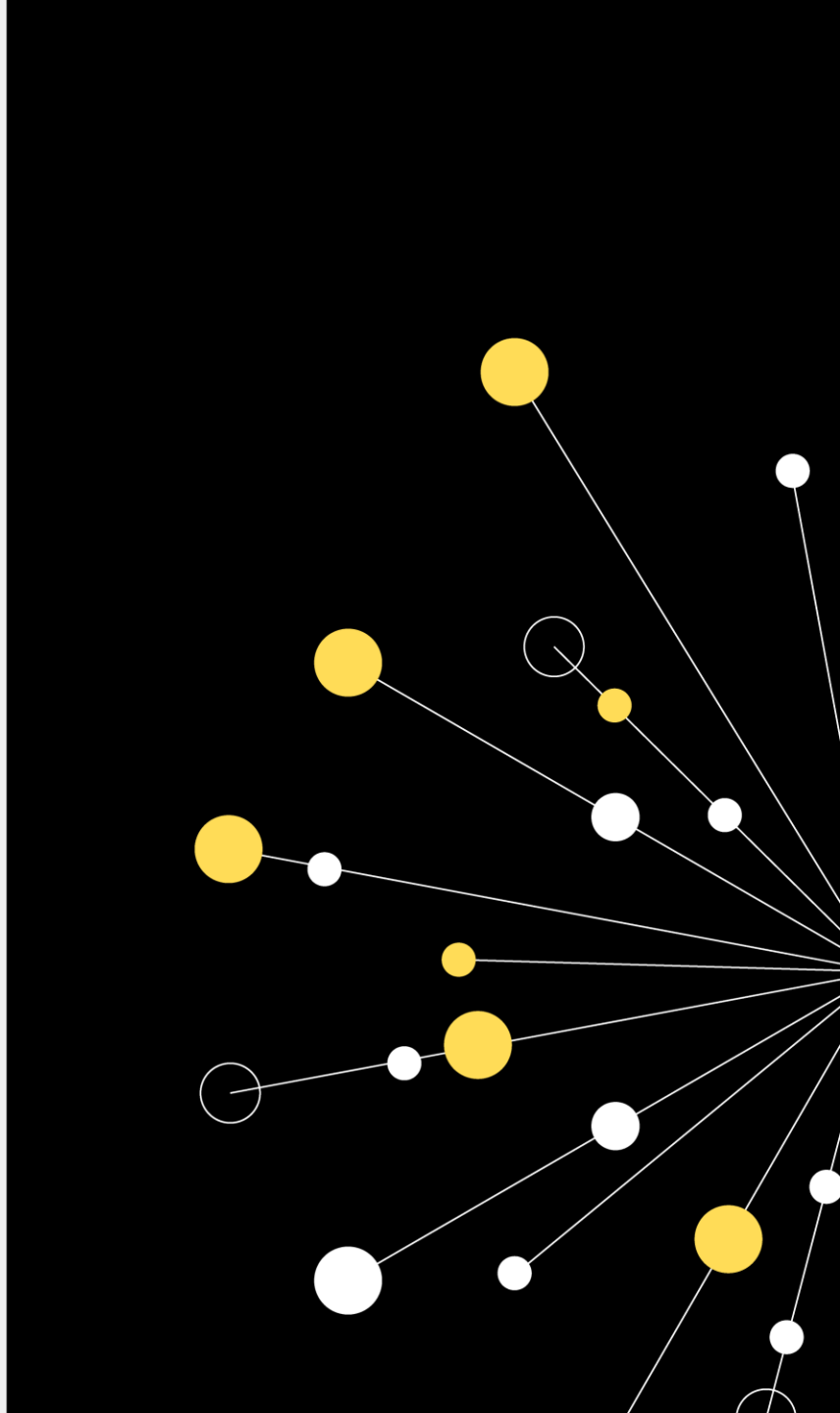
QA

Using Threads

Let's examine manual threading.

```
Thread thread = new Thread(() => {
    Console.WriteLine("Thread started.");
    Thread.Sleep(1000);
});
thread.Start();
thread.Join();
```

Thread gives low-level control but
Tasks are simpler for most cases.



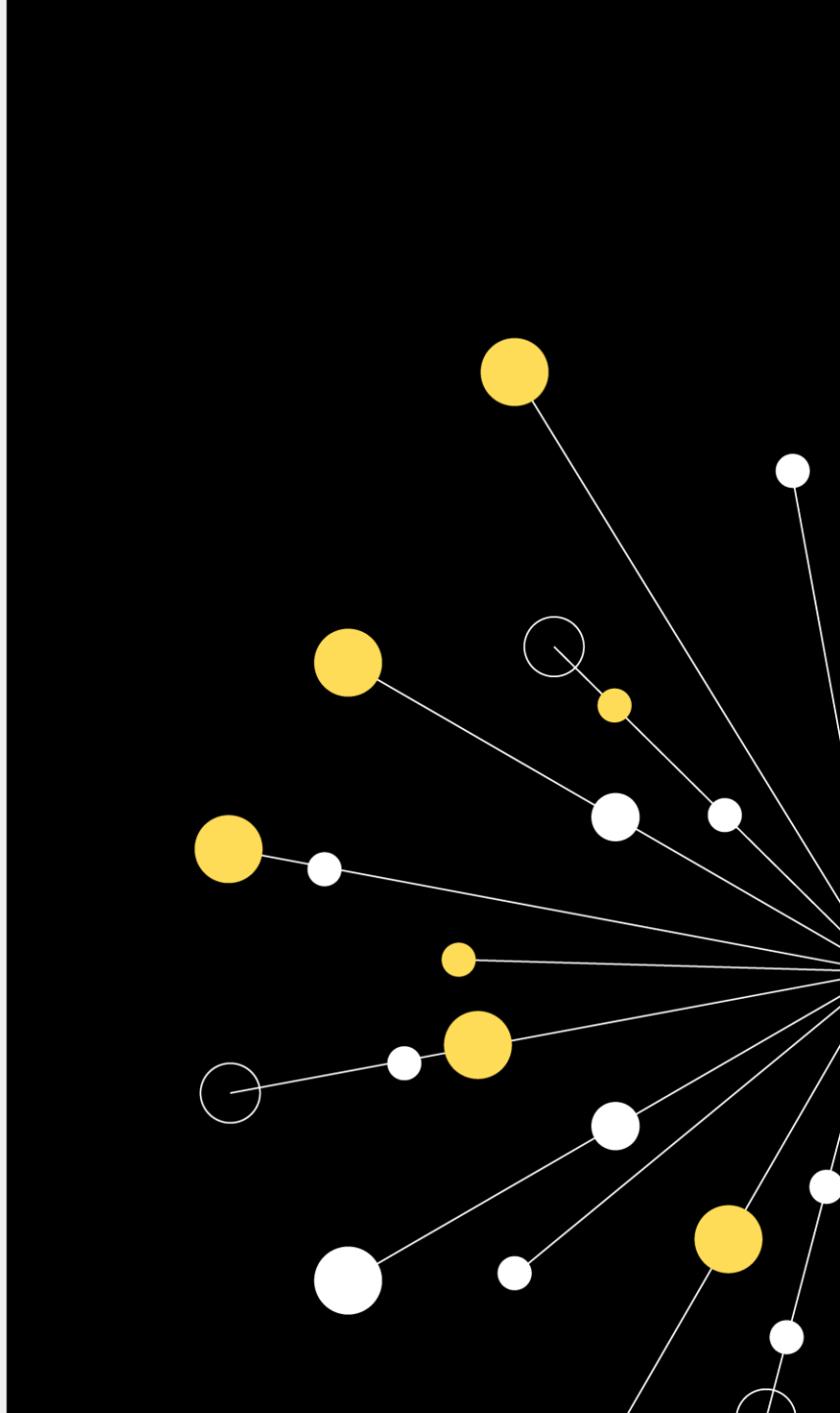
QA

Task Continuations

Chain dependent tasks sequentially.

```
Task.Run(() =>
    Console.WriteLine("Step 1")).
    ContinueWith(t => Console.WriteLine("Step 2")).
    ContinueWith(t => Console.WriteLine("Step 3")).
    Wait();
```

ContinueWith
creates a chain of sequential async operations.



QA

Parallel Loops

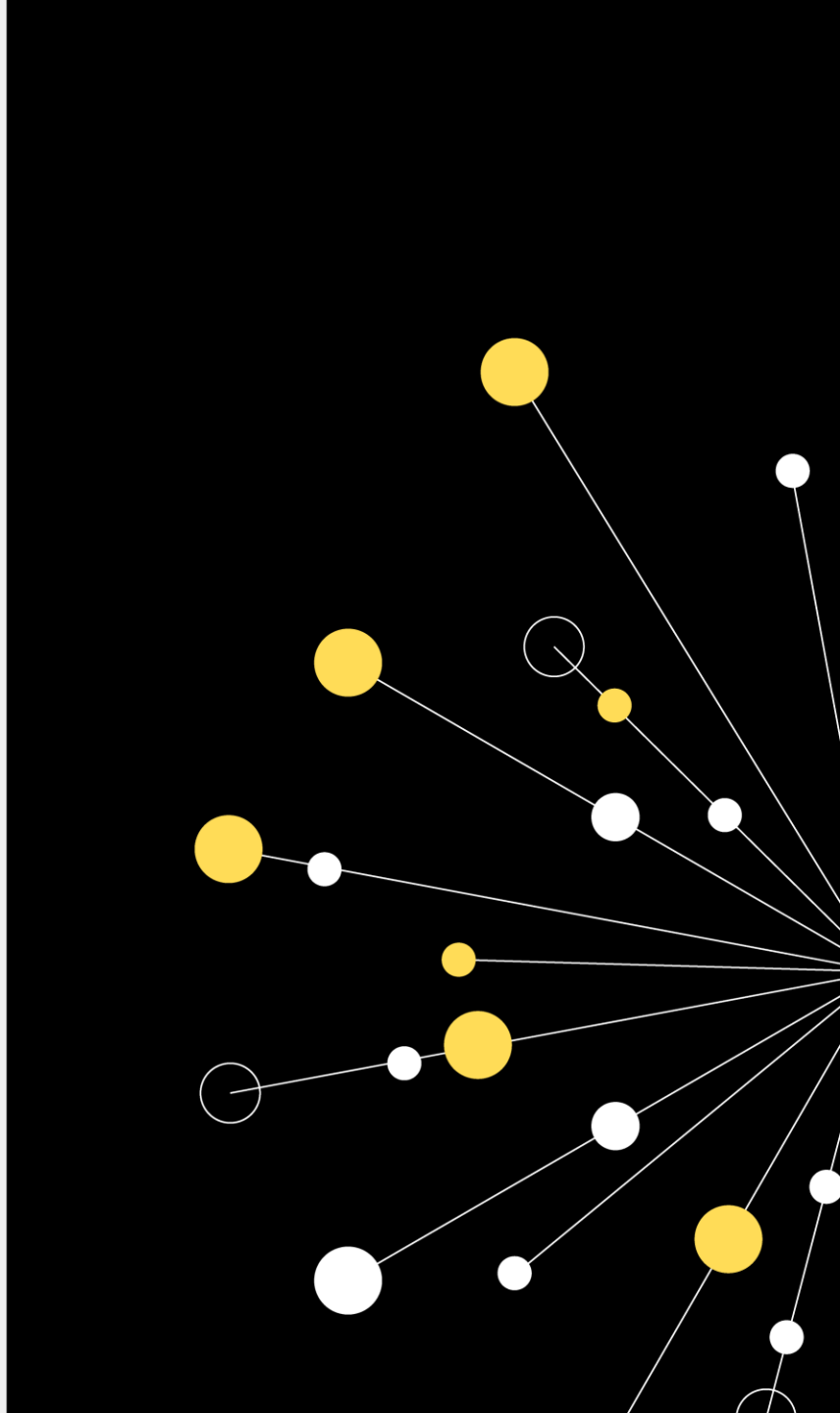
Speed up CPU-bound work with parallel processing.

```
Parallel.For(0, 10, i => {
    Console.WriteLine($"Index {i}, " +
        $"Thread {Task.CurrentId}");
});
```

Executes the body (i => { ... }) for each value of **i from 0 to 9**, but instead of processing them one by one, it **spreads them across multiple threads** from the thread pool.

Parallel.For

Distributes work across threads. Order not guaranteed.



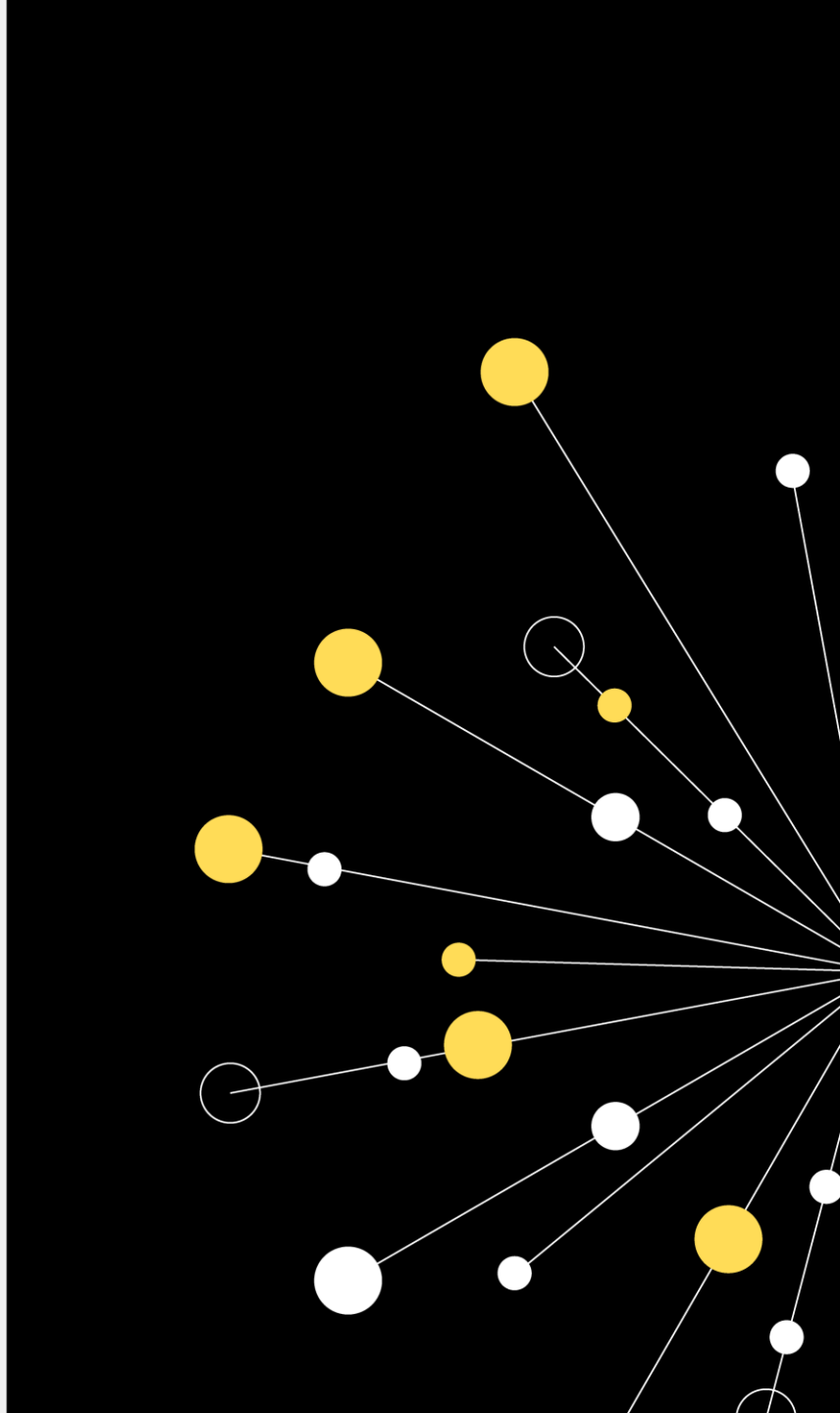
QA

Task Cancellation

Used to stop a running task gracefully using CancellationToken.

```
var cts = new CancellationTokenSource();
Task t = Task.Run(() => {
    for (int i = 0; i < 10; i++) {
        cts.Token.ThrowIfCancellationRequested();
        Task.Delay(500).Wait();
    }
}, cts.Token);
cts.Cancel();
```

CancellationToken allows cooperative stopping of async work.



QA

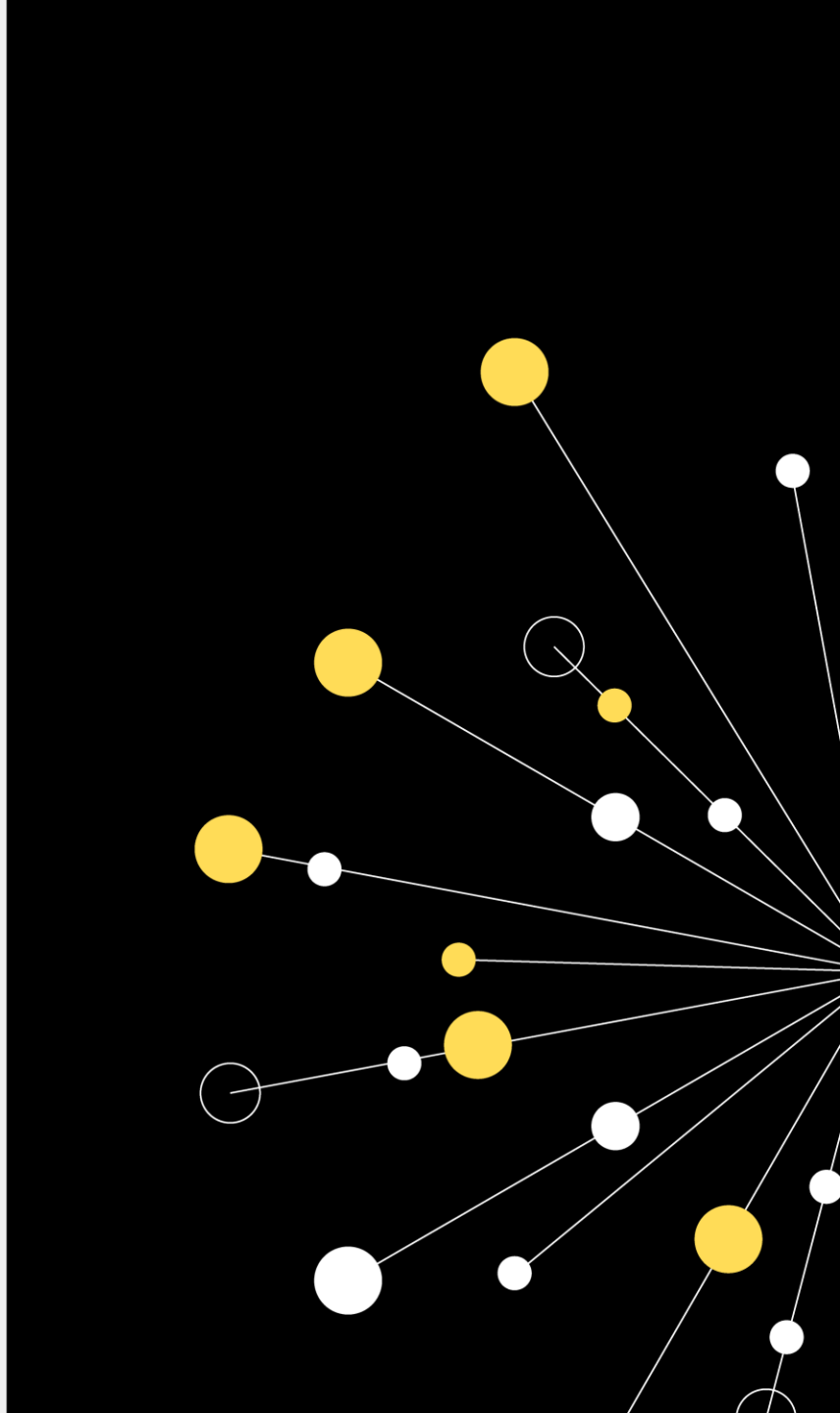
Synchronization (lock)

Goal: Prevent race conditions on shared data.

```
int counter = 0;
object locker = new object();

Parallel.For(0, 1000, i => {
    lock (locker) counter++;
});
```

Lock ensures only one thread modifies data at a time.



QA