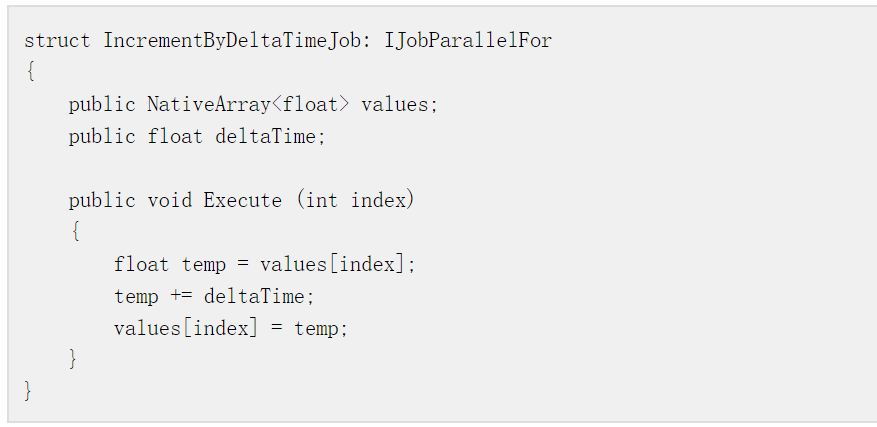
**ParallelFor jobs**

When [scheduling jobs](https://docs.unity3d.com/Manual/JobSystemSchedulingJobs.html), there can only be one job doing one task. In a game, it is common to want to perform the same operation on a large number of objects. There is a separate job type called [IJobParallelFor](https://docs.unity3d.com/ScriptReference/Unity.Jobs.IJobParallelFor.html) to handle this.

**Note**: A “ParallelFor” job is a collective term in Unity for any struct that implements the IJobParallelFor interface.

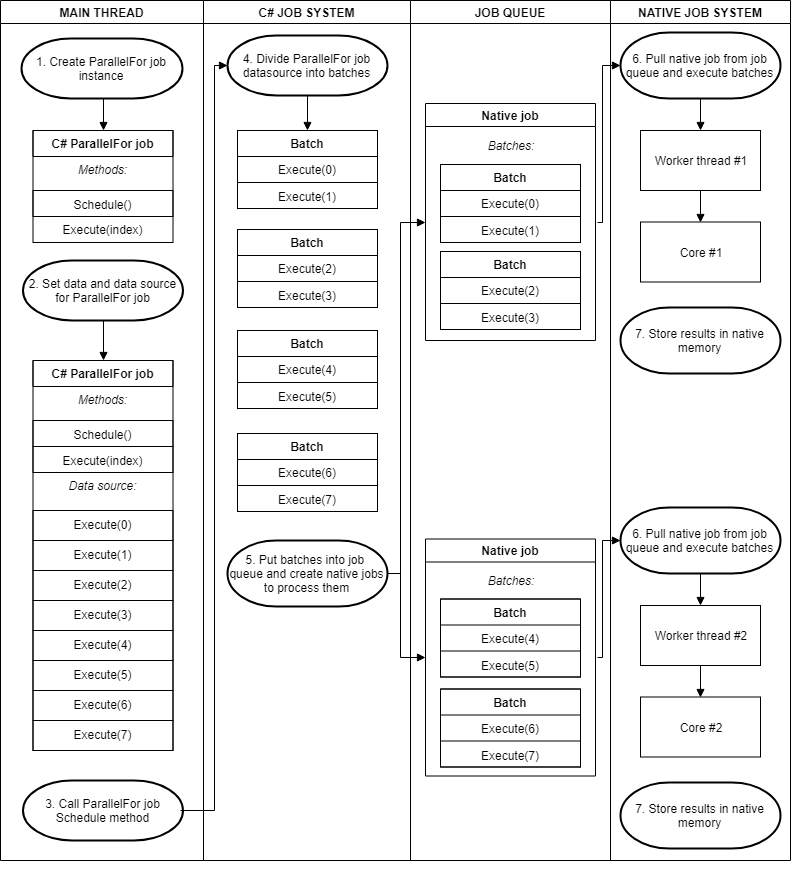
A ParallelFor job uses a [NativeArray](https://docs.unity3d.com/ScriptReference/Unity.Collections.NativeArray_1.html) of data to act on as its data source. ParallelFor jobs run across multiple cores. There is one job per core, each handling a subset of the workload. IJobParallelFor behaves like IJob, but instead of a single [Execute](https://docs.unity3d.com/ScriptReference/Unity.Jobs.IJob.Execute.html) method, it invokes the Execute method once per item in the data source. There is an integer parameter in the Execute method. This index is to access and operate on a single element of the data source within the job implementation.

**An example of a ParallelFor job definition:**

**Scheduling ParallelFor jobs**

When scheduling ParallelFor jobs, you must specify the length of the NativeArray data source that you are splitting. The Unity C# Job System cannot know which NativeArray you want to use as the data source if there are several in the struct. The length also tells the C# Job System how many Execute methods to expect.

Behind the **scenes**, the scheduling of ParallelFor jobs is more complicated. When scheduling ParallelFor jobs, the C# Job System divides the work into batches to distribute between cores. Each batch contains a subset of Execute methods. The C# Job System then schedules up to one job in Unity’s native job system per CPU core and passes that native job some batches to complete.



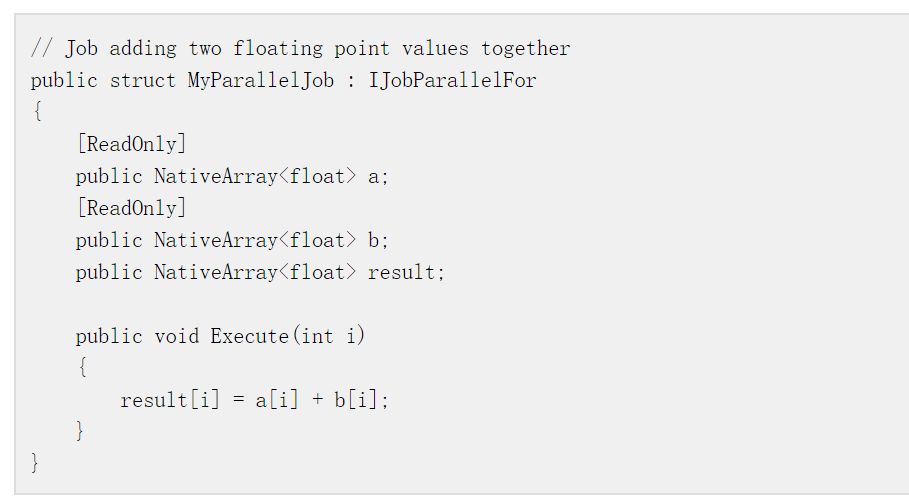
A ParallelFor job dividing batches across cores

When a native job completes its batches before others, it [steals](https://en.wikipedia.org/wiki/Work_stealing) remaining batches from the other native jobs. It only steals half of a native job’s remaining batches at a time, to ensure [cache locality](https://stackoverflow.com/questions/12065774/why-does-cache-locality-matter-for-array-performance).

To optimize the process, you need to specify a batch count. The batch count controls how many jobs you get, and how fine-grained the redistribution of work between threads is. Having a low batch count, such as 1, gives you a more even distribution of work between threads. It does come with some overhead, so sometimes it is better to increase the batch count. Starting at 1 and increasing the batch count until there are negligible performance gains is a valid strategy.

**An example of scheduling a ParallelFor job**

**Job code**:

**Main thread code**:

