This document is best viewed with this option in the menubar: *View → Text Width → Wide*

Unity Job System 101

This document introduces the essential concepts and usage of unmanaged collections and the C# job system. At the end, we'll walk through a simple sample project that demonstrates the basic usage and benefits of jobs.

C# Jobs

The C# job system allows you to put work on a queue to be executed by a pool of worker threads:

- When a worker thread finishes its current work, the thread will pull a waiting job off the queue and run the job (by calling its Execute() method).
- A job type is created by defining a struct that implements IJob or one of the other job interfaces (IJobParallelFor, IJobEntity, IJobChunk...).
- To put a job instance on the job queue, call the extension method Schedule(). Jobs can only be scheduled from the main thread, not from within other jobs.

```
Java
public struct ExampleJob : IJob
{
        // Execute() is called when the job runs.
        public void Execute()
        {
                // ...
        }
}

// ... somewhere on the main thread (such as in a MonoBehaviour update)

// Schedule a job.
var job = new ExampleJob{};
job.Schedule();
```

Job Dependencies

A worker thread will not pull a job off the job queue until the other jobs which it depends upon have all finished execution. Effectively, job dependencies allow you to have newly scheduled jobs wait for previously scheduled jobs and thus control execution order where needed.

A job can be given dependencies only when it is scheduled:

- Schedule() returns a JobHandle representing the new job.
- If a JobHandle is passed to Schedule(), the new job will depend upon the job represented by the handle.

```Java
// Schedule a job.
var job1 = new ExampleJob{};
JobHandle handle1 = job1.Schedule();

// Schedule another job of the same type which depends upon the first.
// Because this job depends upon the first, it will not be executed by
// a worker thread until after the first job finishes execution.
var job2 = new ExampleJob{};
JobHandle handle2 = job1.Schedule(handle1);
```

Although Schedule() only takes one JobHandle argument, we can use JobHandle.CombineDependencies()'s to combine multiple handles into one logical handle, thus allowing us to give a job multiple direct dependencies.

```Java
// Combine three handles into one.
JobHandle combinedHandle = JobHandle.CombineDependencies(handle1, handle2,
handle3);

// Schedule a job that depends directly upon handle1, handle2, and handle3
var job = new ExampleJob{};
job.Schedule(combinedHandle);
```

Completing jobs

At some point after scheduling a job, you should call the JobHandle's Complete() method on the main thread. Completing a job does a few things, in this order:

1. Completes all dependencies of the job (including the dependencies of the dependencies, recursively).
2. Waits for the job to finish execution (if it hasn't finished already).
3. Removes all remaining references to the job from the job queue.

Effectively, once Complete() returns, the job and all its dependencies are guaranteed to have finished execution and to have been removed from the queue.

Also note:

- Calling Complete() on the handle of an already completed job does nothing and throws no error.
- Like with scheduling, only the main thread can complete jobs: calling Complete() within a job is invalid.
- Though a job can be completed immediately after scheduling, it's usually best to hold off completing a job until the latest possible moment when the work actually needs to be finished. In general, the longer the gaps between the scheduling of jobs and their completions, the less likely the main thread and worker threads will spend time needlessly sitting idle.

Data access in jobs

As a general rule:

- A job should not perform I/O.
- A job should not access managed objects.
- A job should only access static fields if they are read only.

Also be clear that scheduling a job creates a *private copy* of the struct that will be visible only to the running job. Consequently, any modifications to the fields in the job will be visible only within the job. However, if a field of the job contains pointers or references, modifications of this external data by the job can be visible outside the job.

Unmanaged collections

The unmanaged collection types of the Unity.Collections package have a few advantages over normal C# managed collections:

- Unmanaged objects can be used in Burst-compiled code.

- Unmanaged objects can be used in jobs (whereas using managed objects in jobs is not normally safe).
- The Native- collection types have “job safety checks” that enforce thread-safety in jobs.
- Unmanaged objects are not garbage collected and so don’t induce garbage collection overhead.

On the downside, you are responsible for calling Dispose() on every unmanaged collection once it's no longer needed. If you neglect to dispose of a collection, this creates a memory leak. For the Native- collections, though, the “disposal safety checks” can catch many (but not all) of these leaks and throw an error.

Allocators

When instantiating an unmanaged collection, you must specify an allocator. Different allocators organize and track their memory in different ways. Three of the most-commonly used allocators are:

- **Allocator.Persistent**: The slowest allocator. Used for indefinite lifetime allocations. You must call Dispose() to deallocate a Persistent-allocated collection when you no longer need it.
- **Allocator.Temp**: The fastest allocator. Used for short-lived allocations. Each frame, the main thread creates a Temp allocator which is deallocated in its entirety at the end of the frame. Because a Temp allocator gets discarded as a whole, you don't actually need to manually deallocate your Temp allocations (and in fact, calling Dispose() on a Temp-allocated collection does nothing).
- **Allocator.TempJob**: Middle-tier speed allocations that can be passed into jobs.

The collections passed to a job must be allocated with Allocator.Persistent, Allocator.TempJob, or another thread-safe allocator.

```Java
public struct ExampleJob : IJob
{
       public NativeList<int> Nums;

       public void Execute() { ... }
}

// ... somewhere on the main thread (such as in a MonoBehaviour update)

// This list we pass to a job must be created with a thread-safe allocator.
// Using Allocator.Temp here instead would trigger an error when we schedule
the job.
var list = new NativeList<int>(100, Allocator.TempJob);

var job = new ExampleJob{ Nums = list };
job.Schedule();
```

Collections allocated with Allocator.Temp cannot be passed into jobs. However, each thread of a job is given its own Temp allocator, so Allocator.Temp is safe to use within jobs. All Temp allocations in a job will be disposed automatically at the end of the job.

Allocations made with Allocator.TempJob must be manually disposed. The disposal safety checks, if enabled, will throw an exception when any allocation made with Allocator.TempJob is not disposed within 4 frames after allocation.

Job Safety Checks

For any two jobs which access the same data, it's generally undesirable for their execution to overlap or for their execution order to be indeterminate. For example, if two jobs read and write the content of a native array, we should ensure that one of the two jobs finishes execution before the other starts. Otherwise, when either job modifies the array, the modifications may interfere with the other job: depending upon the happenstance of which job runs before the other and whether their execution overlaps, one or both jobs might produce the wrong results.

So, when you have such a data conflict between two jobs, you should either:

- Schedule and complete one job before scheduling the other...
- ...or schedule one job as a dependency of the other.

Either way, the two jobs will be guaranteed not to overlap and to execute in a certain order.

When you call Schedule(), the job safety checks (if enabled) will throw an exception if they detect a potential race condition. For instance, an exception will be thrown if you first schedule a job that uses a native array and then schedule a second job which uses that same native array but which does not depend upon the first job. If such an arrangement were allowed, one or both jobs might produce wrong or inconsistent results.

Also:

- As a special case, it's always safe for two jobs to access the same data if both jobs only *read* the data. Because neither job modifies the data, they won't interfere with each other. We can indicate that a native array or collection will only be read in a job by marking the struct field with the [ReadOnly] attribute. The job safety checks will not consider two jobs to conflict if all native arrays or collections they share are marked [ReadOnly] in both jobs.
- In some cases, you may wish to disable the job safety checks entirely for a specific native array or collection used in a job. This can be done by marking it with the [NativeDisableContainerSafetyRestriction] attribute. Just be sure that you're not creating a race condition!
- While a native collection is in use by any currently scheduled jobs, the safety checks will throw an exception if you attempt to read or modify that native collection on the main thread. As a special case, the main thread can read from a native collection if it is marked [ReadOnly] in the scheduled jobs.

Parallel Jobs

To split the work of processing an array or list across multiple threads, we can define a job with the IJobParallelFor interface:

```java
[BurstCompile]
public struct SquareNumbersJob : IJobParallelFor
{
	public NativeArray<int> Nums;

	// Each Execute call processes only a single index.
	public void Execute(int index)
	{
		Nums[index] *= Nums[index];
	}
}
```

When we schedule the job, we specify an index count and batch size:

```java
// ... scheduling the job
```

```
var job = new SquareNumbersJob { Nums = myArray };
JobHandle handle = job.Schedule(
      myArray.Length,     // count
      100);               // batch size
```

When the job runs, its Execute() method will be called *count* times, with all values from 0 up to *count* passed to the *index* param.

The indexes of the job get split into batches determined by the batch size, and the worker threads then can grab these batches individually off the queue. Effectively, the separate batches may be processed concurrently on separate threads, but all indexes of an individual batch will be processed together within a single thread.

In this example, if the array length is, say, 250, then the job will be split into three batches: the first covering indexes 0 through 99; the second covering indexes 100 through 199; and the last batch covering the remainder, indexes 200 through 249. Because the job is split into three batches, it will effectively be processed at most across three worker threads. If we want to split the job up across more threads, we must pick a smaller batch size.

> **Note**: The choice of a good batch size isn't an exact science! In the extreme case, we could pick a batch size of 1 and thereby split each individual index into its own batch. Keep in mind, however, that having too many small batches might incur significant job system overhead. In general, you should pick a batch size that seems not too big but not too small and then experiment to find a size that seems optimal for each specific job.

When a batch is processed, it should only access indexes of an array or list that is within the batch's designated range. To enforce this, the safety checks throw an exception if you access indexes of an array or list with any value other than the index parameter:

Java
```
[BurstCompile]
public struct MyJob : IJobParallelFor
{
      public NativeArray<int> Nums;

      public void Execute(int index)
      {
          // The expression Nums[0] triggers a safety check
      exception!
          Nums[index] = Nums[0];
      }
}
```

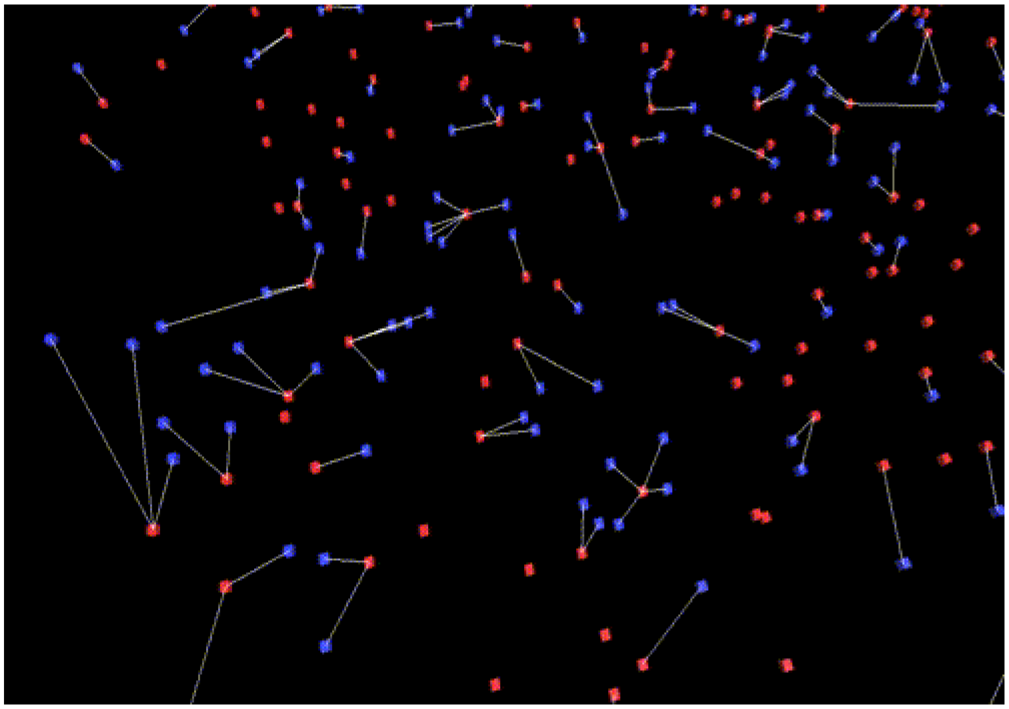
This restriction does not apply to array and list fields marked with the [ReadOnly] attribute. For an array or list that you need to write to in the job, you can disable the restriction by marking the field with [NativeDisableParallelForRestriction]. Just be careful that relaxing this restriction doesn't lead to possible race conditions!

Sample: Targets and Seekers

See the video walkthrough of this sample (17 minutes)

The problem we want to solve:

- Seekers (blue cubes) and Targets (red cubes) each move slowly in a random direction on a 2D plane.
- A white debug line is drawn from each Seeker to its nearest Target.



We'll present four solutions to this problem:

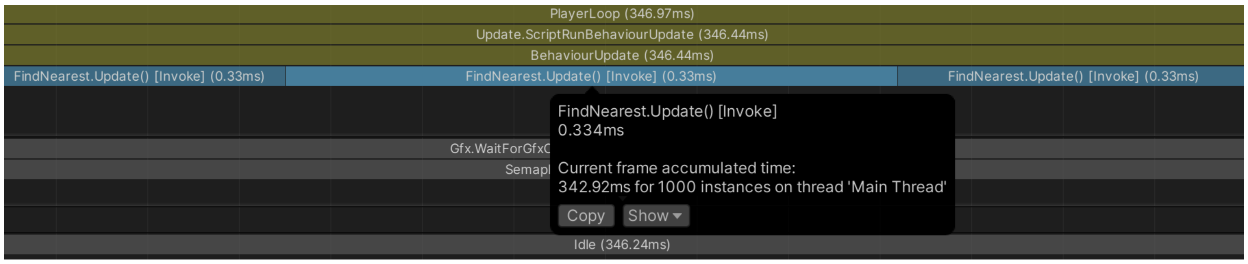
- A solution using no jobs.
- A solution using a single-threaded job.
- A solution using a parallel job.
- And a solution using a parallel job and a smarter-than-brute-force algorithm.

Solution 1 - Main thread only

- A singleton Spawner MonoBehaviour instantiates 1000 seekers (blue cubes) and 1000 targets (red cubes) within a 500-by-500 area on the XZ plane.
- The Seeker and Target MonoBehaviours move each seeker and target slowly along a random vector.
- The Spawner stores the transforms of the targets in a static array.
- The FindNearest MonoBehaviour (attached to the seeker prefab) uses the target transforms array to find the nearest target position and draws a white debug line.

The finding algorithm used here is simple brute force: for every seeker, we loop over every target.

Here's a profile of a typical frame with 1000 seekers and 1000 targets:


PlayerLoop (346.97ms)
Update.ScriptRunBehaviourUpdate (346.44ms)
BehaviourUpdate (346.44ms)
FindNearest.Update() [Invoke] (0.33ms)
FindNearest.Update() [Invoke] (0.33ms)
FindNearest.Update() [Invoke] (0.33ms)
Gfx.WaitForGfxC
Semapl
Copy
Show
Idle (346.24ms)

FindNearest.Update() [Invoke]
0.334ms

Current frame accumulated time:
342.92ms for 1000 instances on thread 'Main Thread'

Each seeker takes ~0.3ms to update, taking over 330ms total.

Solution 2 - Single-threaded job

- The Spawner stores the transforms of both the seekers and targets in static arrays.
- The FindNearest MonoBehaviour is moved from the seeker prefab to the "Spawner" GameObject.
- FindNearest copies the localPosition values of the seeker and target transforms into NativeArrays of float3s.
- FindNearest schedules and completes the new FindNearestJob, which uses these NativeArrays.

By putting the hard work into a job, we can move the work from the main thread to a worker thread, and we can Burst-compile the code. Jobs and Burst-compiled code cannot access any managed objects (including GameObjects and GameObject components), so we must first copy all the data to be processed by the job into unmanaged collections, such as NativeArray's.

> **Note**: Strictly speaking, jobs actually can access managed objects, but doing so requires special care and isn't normally a good idea. Besides, we definitely want to Burst-compile this job, and Burst-compiled code strictly cannot access managed objects at all.

Though we could still use Vector3 and Mathf, we'll instead use float3 and math from the Unity.Mathematics package, which has special optimization hooks for Burst.

So here's how our FindNearestJob job is defined:

Java
```
public struct FindNearestJob : IJob
{
      // All the data which a job will access
      // must be included in its fields.
```

```
        public NativeArray<float3> TargetPositions;
        public NativeArray<float3> SeekerPositions;
        public NativeArray<float3> NearestTargetPositions;

        public void Execute()
        {
                // ... the code which the job will run goes here
        }
}
```

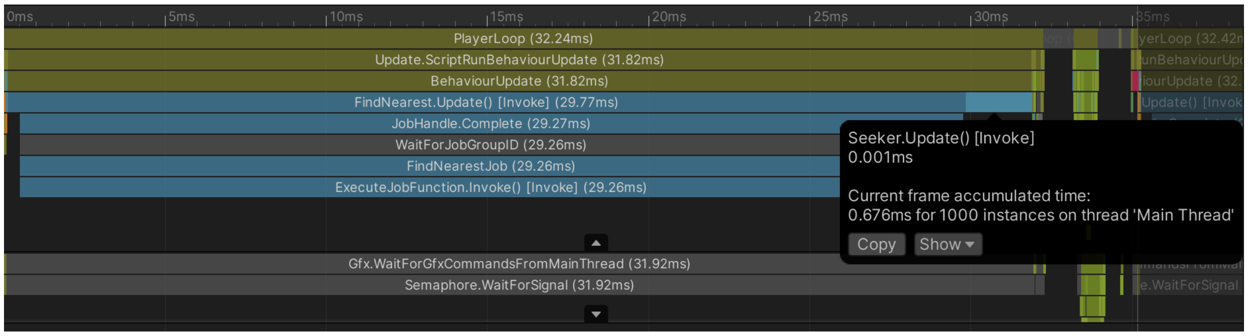
The Update() of FindNearest:

1. Copies the seeker and target transforms into NativeArrays of float3s.
2. Creates an instance of FindNearestJob, initializing its fields with the NativeArrays.
3. Calls the Schedule() extension method on the job instance.
4. Calls Complete() on the JobHandle that was returned by Schedule().
5. Uses the NearestTargetPositions array, which is populated by the job, to draw a debug line from each seeker to its nearest target.

Here's the excerpt that instantiates, schedules, and completes the job:

Java
```
// To schedule a job, we first create an instance and populate
its fields.
FindNearestJob findJob = new FindNearestJob
{
        TargetPositions = TargetPositions,
        SeekerPositions = SeekerPositions,
        NearestTargetPositions = NearestTargetPositions,
};

// Schedule() puts the job instance on the job queue.
JobHandle findHandle = findJob.Schedule();

// The Complete() method will not return until the job
// represented by the handle has finished execution.
// In some cases, a job may have finished
// execution before Complete() is called on its handle.
// Either way, the Complete() call will only return
// once the job is done.
findHandle.Complete();
```

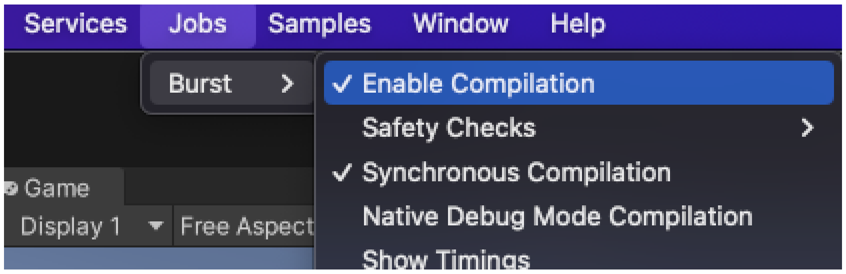
Here's a profile of a typical frame with 1000 seekers and 1000 targets without Burst compiling the job:


0ms
5ms
10ms
15ms
20ms
25ms
30ms
35ms
PlayerLoop (32.24ms)
Update.ScriptRunBehaviourUpdate (31.82ms)
BehaviourUpdate (31.82ms)
FindNearest.Update() [Invoke] (29.77ms)
JobHandle.Complete (29.27ms)
WaitForJobGroupID (29.26ms)
FindNearestJob (29.26ms)
ExecuteJobFunction.Invoke() [Invoke] (29.26ms)
Gfx.WaitForGfxCommandsFromMainThread (31.92ms)
Semaphore.WaitForSignal (31.92ms)
Copy
Show

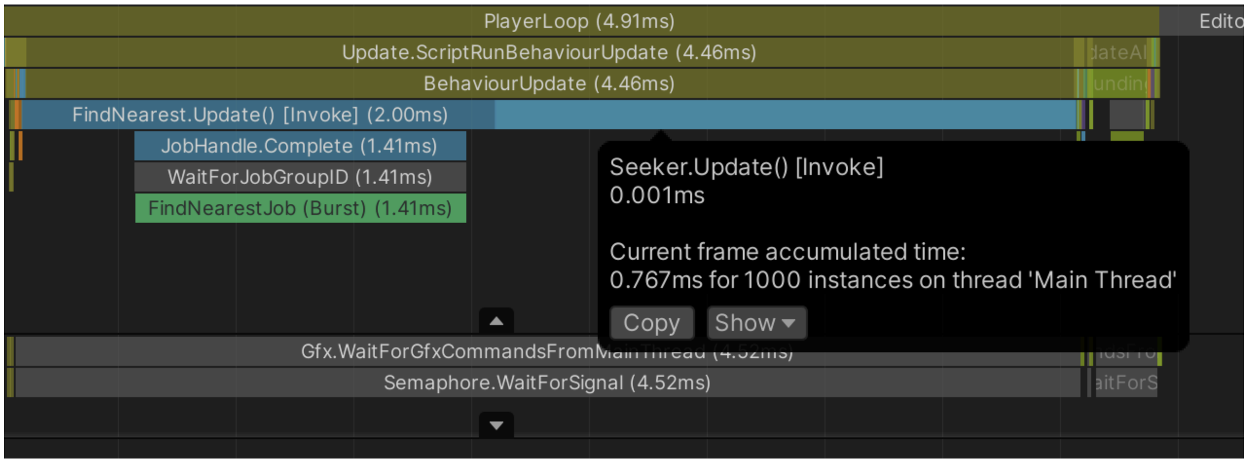
Seeker.Update() [Invoke]
0.001ms

Current frame accumulated time:
0.676ms for 1000 instances on thread 'Main Thread'

~30ms is definitely better than the ~330ms we saw before, but next let's enable Burst compilation by adding the [BurstCompile] attribute on the job struct. Also make sure that Burst compilation is enabled in the menubar:



Here's the result we get with Burst enabled:

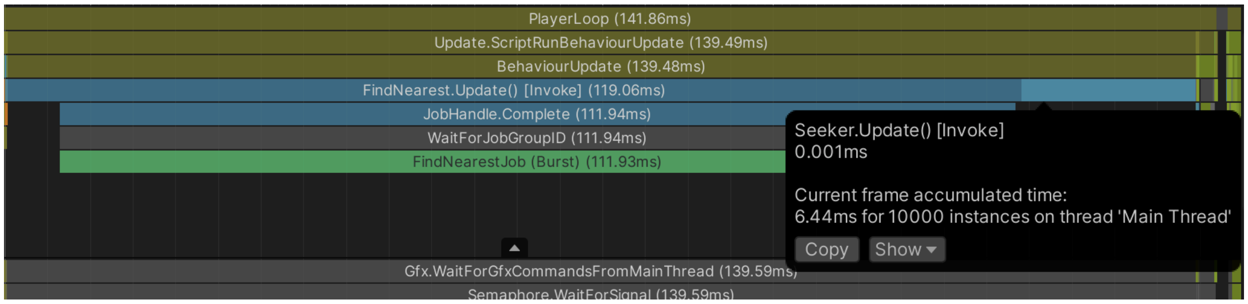

PlayerLoop (4.91ms)
Update.ScriptRunBehaviourUpdate (4.46ms)
BehaviourUpdate (4.46ms)
FindNearest.Update() [Invoke] (2.00ms)
JobHandle.Complete (1.41ms)
WaitForJobGroupID (1.41ms)
FindNearestJob (Burst) (1.41ms)
Gfx.WaitForGfxCommandsFromMainThread (4.52ms)
Semaphore.WaitForSignal (4.52ms)
Copy
Show

Seeker.Update() [Invoke]
0.001ms

Current frame accumulated time:
0.767ms for 1000 instances on thread 'Main Thread'

At ~1.5ms, we're well within the 16.6ms budget of 60fps.

Since we have so much headroom now, let's try increasing to 10,000 seekers and 10,000 targets:


PlayerLoop (141.86ms)
Update.ScriptRunBehaviourUpdate (139.49ms)
BehaviourUpdate (139.48ms)
FindNearest.Update() [Invoke] (119.06ms)
JobHandle.Complete (111.94ms)
WaitForJobGroupID (111.94ms)
FindNearestJob (Burst) (111.93ms)
Gfx.WaitForGfxCommandsFromMainThread (139.59ms)
Semaphore.WaitForSignal (139.59ms)
Copy
Show

Seeker.Update() [Invoke]
0.001ms

Current frame accumulated time:
6.44ms for 10000 instances on thread 'Main Thread'

A 10-fold increase in seekers and targets results in a 70-fold increase in run time, but this is expected given that every seeker is checking its distance to every target.

> **Note**: Notice in these profiles that the jobs run on the main thread. This can happen when we call Complete() on a job that hasn't yet been pulled off the job queue: because the main thread would otherwise just be sitting idle while it waits for the job to finish anyway, the main thread itself might run the job.

Solution 3 - Parallel job

- The FindNearestJob now implements IJobParallelFor instead of IJob.
- The Schedule() call in FindNearest now takes two int parameters: an index count and batch size.

For a job that processes an array or list, it's often possible to parallelize the work by splitting the indices into sub-ranges. For example, one half of an array could be processed on one thread while concurrently the other half is processed on another thread.

We can conveniently create such jobs using IJobParallelFor, whose Schedule() method takes two int arguments:

- index count: the size of the array or list being processed
- batch size: the size of the sub-ranges, a.k.a. batches

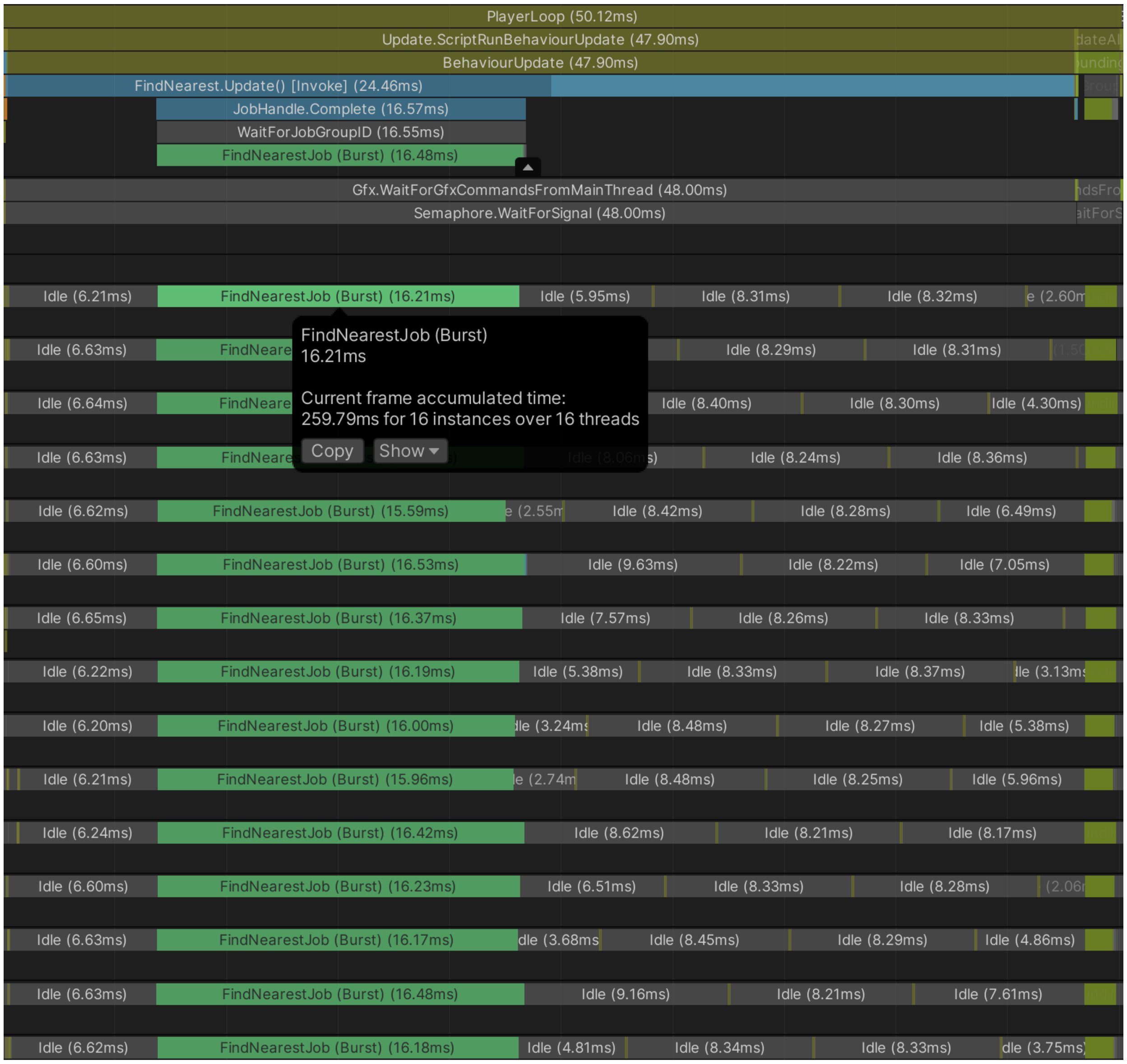
For example, if a job's index count is 100 and its batch size is 40, then the job is split into three batches: the first covering indexes 0 through 39; the second covering indexes 40 through 79; and the third covering indexes 80 through 99.

The worker threads pull these batches off the queue individually, so the batches of a single job can be processed concurrently on different threads.

An IJobParallelFor's Execute() method takes an index parameter and is called once for each index, from 0 up to the index count:

```Java
public struct FindNearestJob : IJobParallelFor
{
      [ReadOnly] public NativeArray<float3> TargetPositions;
      [ReadOnly] public NativeArray<float3> SeekerPositions;
      public NativeArray<float3> NearestTargetPositions;

      // Each Execute call processes only an individual index.
      public void Execute(int index)
      {
            // ...
      }
}

// This job processes every seeker, so the
// seeker array length is used as the index count.
// A batch size of 100 is semi-arbitrarily chosen here
// simply because it's not too big but not too small.
JobHandle findHandle = findJob.Schedule(SeekerPositions.Length,
100);
```

Profile of a typical frame with 10,000 seekers and 10,000 targets:



With the work split across 16 cores, it takes ~260ms total CPU time but less than 17ms elapsed time from start to end.

Solution 4 - Parallel job and a smarter algorithm

- The FindNearest MonoBehaviour now schedules an additional job to sort the target positions array based on the X coords.

- Because the target positions array is sorted, FindNearestJob no longer has to exhaustively consider every target.

We can get big performance gains if we organize the data in some way, such as by sorting the targets into a quadtree or k-d tree. To keep things simple, we'll just sort the targets by their X coords, though we could just as well sort by Z coords (the choice is arbitrary). Finding the nearest target to an individual seeker can then be done in these steps:

1. Do a binary search for the target with the X coord that is nearest to the seeker's X coord.
2. From the index of that target, search up and down in the array for a target with a smaller 2-dimensional distance.
3. As we search up and down through the array, we early out once the X-axis distance exceeds the 2-dimensional distance to the current candidate for nearest target.

The key insights here are that:

- We don't need to check any target with an X-axis distance greater than the 2-dimensional distance to the current candidate.
- Assuming the targets are sorted by their X coord, if the X-axis distance of one target is too great, then the X-axis distances of all targets to one side of it in the array must also be too large.

So the search no longer requires considering every individual target for each seeker.

We could write our own job to do the sorting, but instead we'll call the NativeArray extension method SortJob(), which returns a SortJob struct. The Schedule() method of SortJob schedules two jobs: SegmentSort, which sorts separate segments of the array in parallel, and SegmentSortMerge, which merges the sorted segments. (Merging must be done in a separate single-threaded job because the merging algorithm can't be parallelized.)

The SegmentSortMerge job should not begin execution until the SegmentSort job has itself finished execution, so the SegmentSort job is made a dependency of the SegmentSortMerge job. As a rule, the worker threads will not execute a job until all of its dependencies have finished execution. Job dependencies effectively allow us to specify a sequential execution order amongst the jobs we schedule.

Our FindNearestJob needs to wait for the sorting to finish, so it must depend upon the sorting jobs. Here's our code that creates and schedules the jobs:

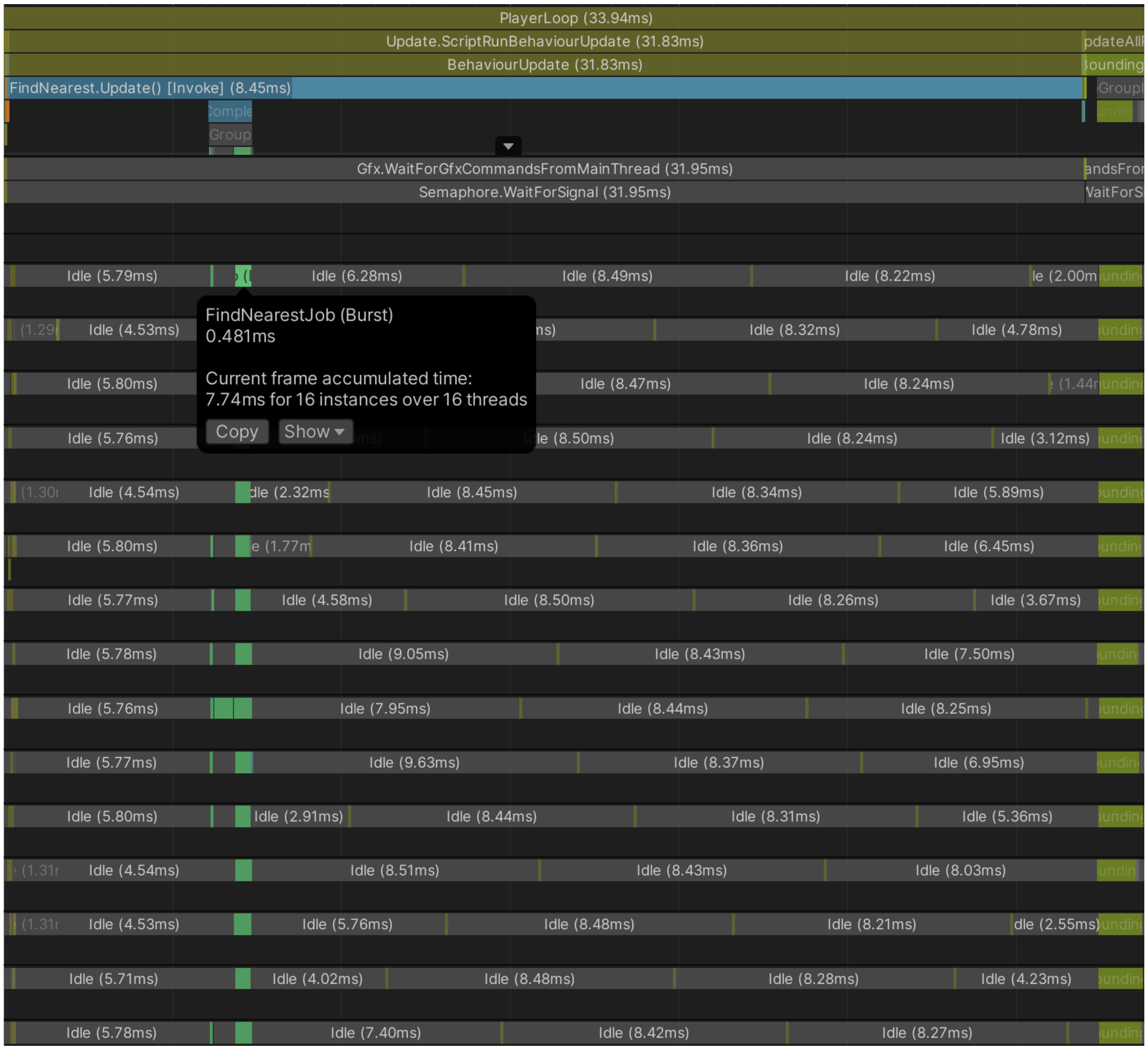
```java
SortJob<float3, AxisXComparer> sortJob = TargetPositions.SortJob(
    new AxisXComparer { });
```

```
FindNearestJob findJob = new FindNearestJob
{
      TargetPositions = TargetPositions,
      SeekerPositions = SeekerPositions,
      NearestTargetPositions = NearestTargetPositions,
};

JobHandle sortHandle = sortJob.Schedule();

// To make the find job depend upon the sorting jobs,
// we pass the sort job handle when we schedule the find job.
JobHandle findHandle = findJob.Schedule(
      SeekerPositions.Length, 100, sortHandle);

// Completing a job also completes all of its
// dependencies, so completing the find job also
// completes the sort jobs.
findHandle.Complete();
```

The sequence of jobs is thus: SegmentSort -> SegmentSortMerge -> FindNearestJob.

If we neglect to make the sorting job a dependency of the find job, the job safety checks will throw an exception when we attempt to schedule the find job.

For this solution, here's a profile of a typical frame with 10,000 seekers and 10,000 targets:

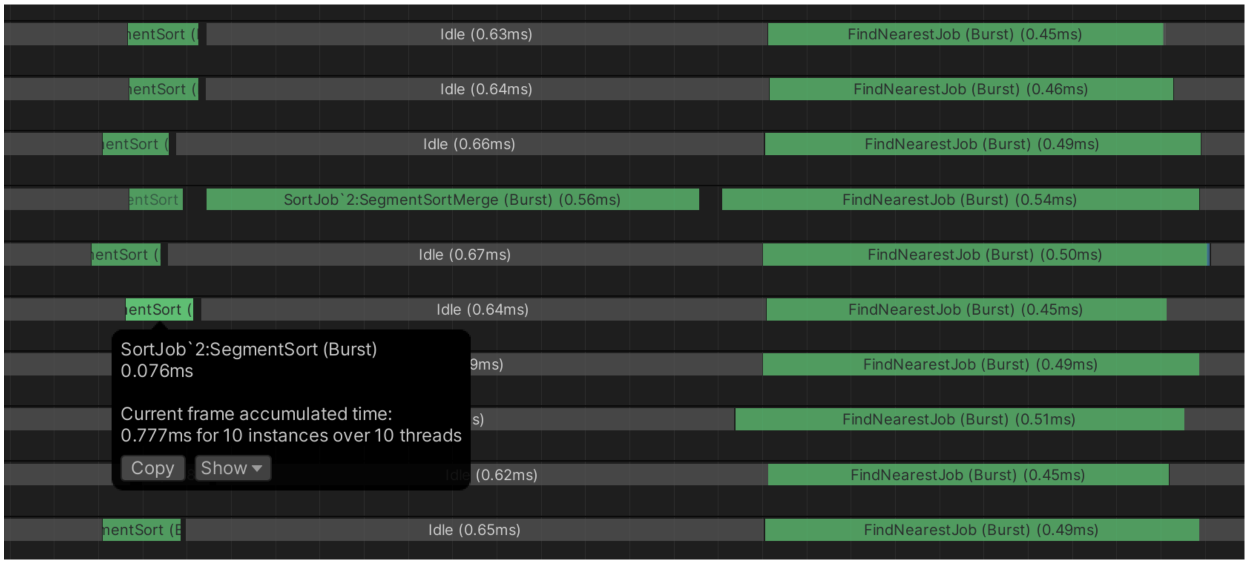

PlayerLoop (33.94ms)
Update.ScriptRunBehaviourUpdate (31.83ms)
BehaviourUpdate (31.83ms)
FindNearest.Update() [Invoke] (8.45ms)
Gfx.WaitForGfxCommandsFromMainThread (31.95ms)
Semaphore.WaitForSignal (31.95ms)
Idle (5.79ms) Idle (6.28ms) Idle (8.49ms) Idle (8.22ms)
Idle (4.53ms) Idle (8.32ms) Idle (4.78ms)
Idle (5.80ms) Idle (8.47ms) Idle (8.24ms)
Idle (5.76ms) Idle (8.50ms) Idle (8.24ms) Idle (3.12ms)
Idle (4.54ms) Idle (8.45ms) Idle (8.34ms) Idle (5.89ms)
Idle (5.80ms) Idle (8.41ms) Idle (8.36ms) Idle (6.45ms)
Idle (5.77ms) Idle (4.58ms) Idle (8.50ms) Idle (8.26ms) Idle (3.67ms)
Idle (5.78ms) Idle (9.05ms) Idle (8.43ms) Idle (7.50ms)
Idle (5.76ms) Idle (7.95ms) Idle (8.44ms) Idle (8.25ms)
Idle (5.77ms) Idle (9.63ms) Idle (8.37ms) Idle (6.95ms)
Idle (5.80ms) Idle (2.91ms) Idle (8.44ms) Idle (8.31ms) Idle (5.36ms)
Idle (4.54ms) Idle (8.51ms) Idle (8.43ms) Idle (8.03ms)
Idle (4.53ms) Idle (5.76ms) Idle (8.48ms) Idle (8.21ms)
Idle (5.71ms) Idle (4.02ms) Idle (8.48ms) Idle (8.28ms) Idle (4.23ms)
Idle (5.78ms) Idle (7.40ms) Idle (8.42ms) Idle (8.27ms)
Copy
Show

FindNearestJob (Burst)
0.481ms

Current frame accumulated time:
7.74ms for 16 instances over 16 threads

Now the FindNearestJob takes just ~7.5ms total CPU time and ~0.5ms elapsed time from start to end.

Zooming in, we can see the SegmentSort and SegmentSortMerge jobs:


entSort (
Idle (0.63ms)
FindNearestJob (Burst) (0.45ms)
entSort (
Idle (0.64ms)
FindNearestJob (Burst) (0.46ms)
entSort (
Idle (0.66ms)
FindNearestJob (Burst) (0.49ms)
entSort
SortJob`2:SegmentSortMerge (Burst) (0.56ms)
FindNearestJob (Burst) (0.54ms)
entSort (
Idle (0.67ms)
FindNearestJob (Burst) (0.50ms)
entSort (
Idle (0.64ms)
FindNearestJob (Burst) (0.45ms)
9ms)
FindNearestJob (Burst) (0.49ms)
s)
FindNearestJob (Burst) (0.51ms)
(0.62ms)
FindNearestJob (Burst) (0.45ms)
entSort (E
Idle (0.65ms)
FindNearestJob (Burst) (0.49ms)
Copy
Show

```
SortJob`2:SegmentSort (Burst)
0.076ms

Current frame accumulated time:
0.777ms for 10 instances over 10 threads
```

The SegmentSort takes under a 0.1ms start to end, and the single-threaded SegmentSortMerge takes ~0.5ms. Weighed against the enormous improvement in FindNearestJob, the extra step of sorting is well worth the additional cost.

Most of the frame time now is eaten up by the inefficiencies of GameObjects, which could be addressed by replacing the GameObjects with entities.

Additional resources

- Collections package cheatsheet
- Blog post: Improving Job System Performance part 1
- Blog post: Improving Job System Performance part 2