



Internal Multithreading

Easily harness multicore processors to speed up your simulation.

1 | Getting Started

BepuPhysics supports the usage of an arbitrary number of threads. Modern PC's with multicore processors and the Xbox360 get a significant performance boost in over the sequential alternative.

The process of getting multithreading working is very simple;

1. threads are given to the engine, and
2. the engine is told that it can use the threads.

1.A | The ThreadManager

The engine must first be told to use extra threads. This is where the space's ThreadManager property comes in; it offers an easy "AddThread" method that can be called to tell the engine to use another thread. As a user, no management of a Thread object is necessary. The thread is created and handled internally. Every Space object has its own ThreadManager property.

When adding a thread, an initializing delegate can be specified. Usually, this is used on the Xbox360 to call Thread.CurrentThread.SetProcessorAffinity; more information about this can be found in section 1.D.

Once the threads are added, set the space's UseMultithreadedUpdate property to true. Multithreading should only be activated if more than one thread has been added to the ThreadManager as the extra overhead involved in allocating work to threads would make it somewhat slower than the sequential version.

The ThreadManager also offers arbitrary task enqueueing, waiting on the tasks' completion, and doing parallel for loops. These are used internally to parallelize the engine, but can also be used externally (as in custom multithreaded updateables, described later).

1.B | Work Distribution

Each stage of the engine must wait for all threads to complete their task before continuing, so spreading and balancing the load is beneficial. Ideally, the ThreadManager will be given the same number of threads as the computer has free processors.

A Space will only use the threads of the ThreadManager while inside its update method. This means, after the

update method, all of the processor power is free to do other things. In an otherwise sequential game, the engine can afford to use every processor. If the space is being updated asynchronously while some other components need processor power, care must be taken to ensure that bottlenecks are not being created.

1.C | Windows Threading

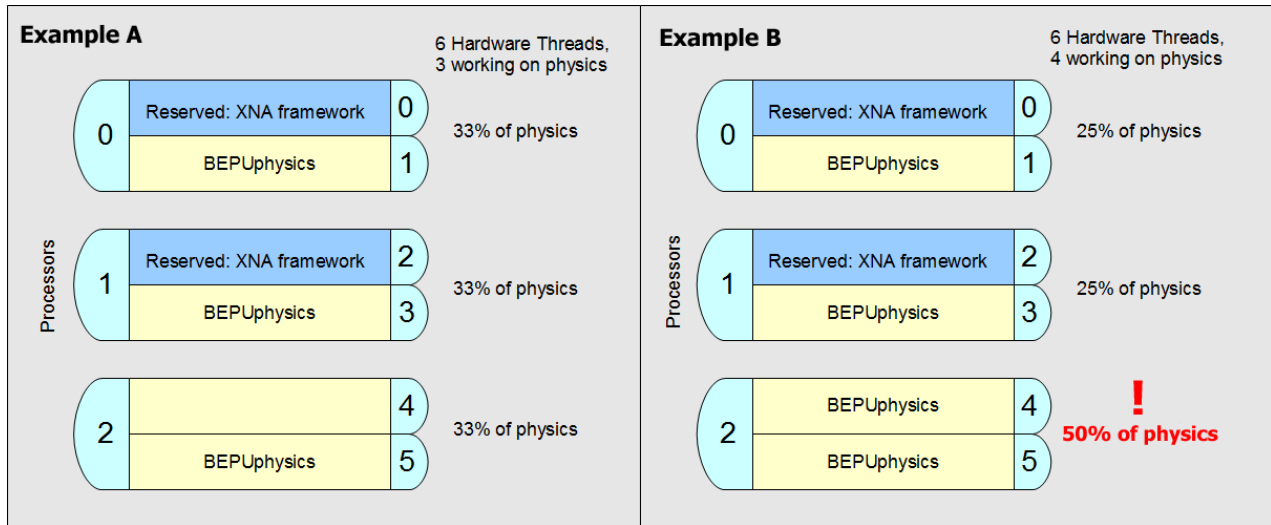
The thread scheduler on Windows can effectively determine which processor should work on a thread. The user can check how many processors/hardware threads are available by using the .NET framework's `System.Environment.ProcessorCount` property. Adding as many threads as processors exist generally works well, assuming there is more than one processor.

The PC implementation of the `IThreadManager` is also capable of automatically balancing parallel for loops, helping it play better with other processes and CPU loads.

1.D | Xbox360 Threading

On the Xbox360, threads should be given a particular location to run. Out of the Xbox's six available hardware threads, 0 and 2 are reserved by the XNA framework, leaving hardware threads 1, 3, 4 and 5 for other work. The thread affinity of a thread being added to the `ThreadManager` can be specified by giving the `ThreadManager's AddThread` method a delegate which sets the affinity using `Thread.CurrentThread.SetProcessorAffinity`.

There are two hardware threads per core on the Xbox, and both hardware threads 4 and 5 are on the same core. The Xbox implementation of the `IThreadManager` does not balance loads, so to have an equal distribution of work amongst the cores, it is a good idea to only use three of the hardware threads (1, 3, and 5).



Using less than four hardware threads is important because the Xbox `IThreadManager` implementation splits the workload up evenly amongst the available threads. While the Xbox's hardware threads do execute software threads independently, execution will slow down on one hardware thread if the core's other hardware thread is

busy. The first two cores will likely finish early and sit idle while the last core tries to finish its extra work, eliminating any benefit to using the last core's second hardware thread.

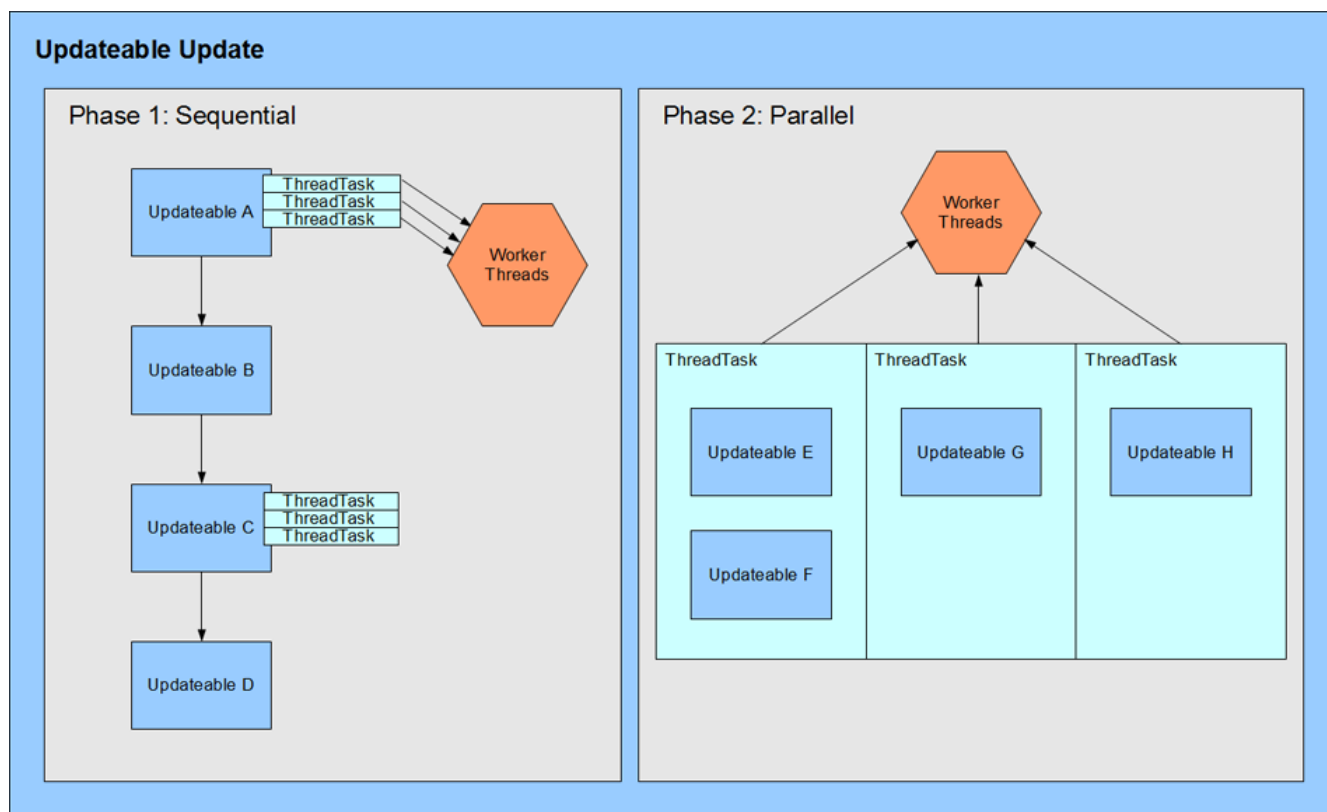
2 | Extensible Types and Threading

Updateables and SolverUpdateables both interact with threading to provide options for speeding up custom types.

2.A | Updateable Threading

Updateables are a method for extending BEPUphysics. Each updateable has multiple stages which are called automatically at various points in the execution of the space's update. Updateables can work in three different ways with respect to threading:

- Updateable does make use of worker threads and executes sequentially
- Updateable does not make use of worker threads and executes sequentially
- Updateable does not make use of worker threads and executes in parallel with other Updateables



The default style is to use no threads and execute sequentially for simplicity. The style can be changed by setting an Updateable's `IsUpdatedSequentially` property. When an Updateable is updated sequentially, no other Updateables or BEPUphysics-related tasks are running at the same time and all of the threads of the ThreadManager are usable. This is helpful for speeding up a large, performance-critical Updateable.

If `IsUpdatedSequentially` is false, the Updateable will run side-by-side with other Updateables that also have `IsUpdatedSequentially` set to false. The ThreadManager should not be used by these Updateables as the ThreadManager's threads are executing the Updateables themselves. Since these Updateables are running in parallel, they must operate in a thread-safe way.

2.B | SolverUpdateable Threading

SolverUpdateables are used in the velocity solver to manage velocity-based interactions, like collisions. They have two important methods: `PreStep` and `ApplyImpulse`. The `PreStep` method performs any per-frame calculations while the `ApplyImpulse` method converges to the desired solution by being called repeatedly. Every SolverUpdateable affects a set of entities, stored in an `involvedEntities` list.

While `PreStep` and `ApplyImpulse` are executing, they should have exclusive access to all entities within the `involvedEntities` list. This is accomplished in different ways depending on what kind of solver type is being used.

When a space's `UseBatchedMultithreadingSolver` property is false (as it is by default), a lock is acquired on each involved entity's locker object before entering either the `PreStep` or `ApplyImpulse` method. Upon completing the method, the locks are released.

If a space's `UseBatchedMultithreadingSolver` property is true, no explicit locking is performed around the `ApplyImpulse` method. Instead, only non-conflicting SolverUpdateables are ever solved in parallel. The `PreStep` methods are still locked. This method is not used by default due to higher overhead associated with the batching. With very large loads and core counts, this solver approach can be more appealing.

If the `involvedEntities` list needs to be changed in the `PreStep` method, the lock can be temporarily released by calling the SolverUpdateable's `ExitLock` function. After the `involvedEntities` list is modified, `EnterLock` can be called to re-acquire the lock.

3 | Miscellaneous Details

There are a variety of other details to take into account when delving deeply into the multithreading system.

3.A | Determinism

The default multithreading system is **not** deterministic. If the same simulation is run multiple times with exactly the same starting conditions, the result can be different. The longer the simulations run, the more they will diverge.

For example, when creating a networking system that handles physics, the engine should not be relied upon to generate consistent results on the various clients even if the input parameters are all identical. Periodic synchronizations are required to keep everything running consistently.

The sources of this nondeterminism are the multithreaded `DynamicBinaryHierarchy` broad phase and the non-batched multithreaded solver. The order of pairs generated by the `DynamicBinaryHierarchy` is unpredictable, leading to different solving orders and slightly different results. Similarly, the solving order of the non-batched solver is based on how fast threads can work through the list and will produce slightly different results.

Every `BroadPhase` has an `AllowMultithreading` field. Setting this to false will force the broad phase to use the sequential (deterministic) version.

The solver can be made deterministic in two ways. It can be made sequential by setting the space's `allowMultithreadedSolver` field to false, or it can be changed to a batched mode which is deterministic. The batched mode can be enabled by setting the space's `UseBatchedMultithreadingSolver` to true. Using the batched solver will incur a significant overhead on some simulations, though it can technically scale to manycore systems better than the default solver in some large simulations.

In general, it is recommended that the normal systems are used (non-batched multithreading solver with full broadphase multithreading) and that determinism should not be designed to be a requirement for a simulation.

3.B | Thread-specific Resource Pools

`ResourcePools` provide a clean and quick way to access commonly used resource types. Since the resource pool is accessed from many different threads, lock contention can become an issue. On the PC, this is helped by using thread-specific pools in the `Resources` static class.

Since the pools are specific to each thread, it is important to ensure that a pool never becomes 'starved' with resources constantly requested, but none ever returned. Starvation can occur in the following sequence:

1. Resource requested by the main thread
2. Resource sent to worker thread
3. Resource returned to pool by worker thread

The main thread's resource pool will be emptied quickly while the worker threads' pools will continually grow. To avoid this, try to always return a resource to the pool from the same thread the resource was requested on.

The two built-in types of `ResourcePool` in the engine are the `UnsafeResourcePool` and `LockingResourcePool`. `UnsafeResourcePools` are low-overhead, direct access pools, but are not thread-safe (and are used in the thread static pools above). `LockingResourcePools` perform a busy-lock to manage synchronization.

3.C | IThreadManager Types

The engine provides a SimpleThreadManager, ThreadTaskManager, and SpecializedThreadManager. On the PC, the SpecializedThreadManager is used, while on the Xbox360, the ThreadTaskManager is used. The SimpleThreadManager is a reference implementation from v0.10.0.

Custom IThreadManager implementations can be created and used. The XNA 4.0 version of the library provides an experimental TPL-based thread manager, though its performance is currently slightly worse than that of the SpecializedThreadManager on the PC.