When to use a thread-safe collection

Article • 09/15/2021 • 3 minutes to read

.NET Framework 4 introduced five collection types that are specially designed to support multi-threaded add and remove operations. To achieve thread-safety, these types use various kinds of efficient locking and lock-free synchronization mechanisms. Synchronization adds overhead to an operation. The amount of overhead depends on the kind of synchronization that is used, the kind of operations that are performed, and other factors such as the number of threads that are trying to concurrently access the collection.

In some scenarios, synchronization overhead is negligible and enables the multi-threaded type to perform significantly faster and scale far better than its non-thread-safe equivalent when protected by an external lock. In other scenarios, the overhead can cause the thread-safe type to perform and scale about the same or even more slowly than the externally-locked, non-thread-safe version of the type.

The following sections provide general guidance about when to use a thread-safe collection versus its non-thread-safe equivalent that has a user-provided lock around its read and write operations. Because performance may vary depending on many factors, the guidance is not specific and is not necessarily valid in all circumstances. If performance is very important, then the best way to determine which collection type to use is to measure performance based on representative computer configurations and loads. This document uses the following terms:

Pure producer-consumer scenario
Any given thread is either adding or removing elements, but not both.

Mixed producer-consumer scenario
Any given thread is both adding and removing elements.

Speedup
Faster algorithmic performance relative to another type in the same scenario.

Scalability
The increase in performance that is proportional to the number of cores on the computer. An algorithm that scales performs faster on eight cores than it does on two cores.

ConcurrentQueue(T) vs. Queue(T)

In pure producer-consumer scenarios, where the processing time for each element is very small (a few instructions), then System.Collections.Concurrent.ConcurrentQueue<T> can offer modest performance benefits over a System.Collections.Generic.Queue<T> that has an external lock. In this scenario, ConcurrentQueue<T> performs best when one dedicated thread is queuing and one dedicated thread is de-queuing. If you do not enforce this rule, then Queue<T> might even perform slightly faster than ConcurrentQueue<T> on computers that have multiple cores.

When processing time is around 500 FLOPS (floating point operations) or more, then the two-thread rule does not apply to ConcurrentQueue<T>, which then has very good scalability. Queue<T> does not scale well in this scenario.

In mixed producer-consumer scenarios, when the processing time is very small, a Queue<T> that has an external lock scales better than ConcurrentQueue<T> does. However, when processing time is around 500 FLOPS or more, then the ConcurrentQueue<T> scales better.

ConcurrentStack vs. Stack

In pure producer-consumer scenarios, when processing time is very small, then System.Collections.Concurrent.ConcurrentStack<T> and System.Collections.Generic.Stack<T> that has an external lock will probably perform about the same with one dedicated pushing thread and one dedicated popping thread. However, as the number of threads increases, both types slow down because of increased contention, and Stack<T> might perform better than ConcurrentStack<T>. When processing time is around 500 FLOPS or more, then both types scale at about the same rate.

In mixed producer-consumer scenarios, ConcurrentStack<T> is faster for both small and large workloads.

The use of the PushRange and TryPopRange may greatly speed up access times.

ConcurrentDictionary vs. Dictionary

In general, use a System.Collections.Concurrent.ConcurrentDictionary<TKey,TValue> in any scenario where you are adding and updating keys or values concurrently from multiple threads. In scenarios that involve frequent updates and relatively few reads, the ConcurrentDictionary<TKey,TValue> generally offers modest benefits. In scenarios that involve many reads and many updates, the ConcurrentDictionary<TKey,TValue> generally is significantly faster on computers that have any number of cores.

In scenarios that involve frequent updates, you can increase the degree of concurrency in the ConcurrentDictionary<TKey,TValue> and then measure to see whether performance increases on computers that have more cores. If you change the concurrency level, avoid global operations as much as possible.

If you are only reading key or values, the Dictionary<TKey,TValue> is faster because no synchronization is required if the dictionary is not being modified by any threads.

ConcurrentBag

In pure producer-consumer scenarios, System.Collections.Concurrent.ConcurrentBag<T> will probably perform more slowly than the other concurrent collection types.

In mixed producer-consumer scenarios, ConcurrentBag<T> is generally much faster and more scalable than any other concurrent collection type for both large and small workloads.

BlockingCollection

When bounding and blocking semantics are required, System.Collections.Concurrent.BlockingCollection<T> will probably perform faster than any custom implementation. It also supports rich cancellation, enumeration, and exception handling.

See also

- System.Collections.Concurrent
- Thread-Safe Collections
- Parallel Programming