Thread-safe Collections in .NET Framework 4 and Their Performance Characteristics

Chunyan Song, Emad Omara, Mike Liddell
Parallel Computing Platform
Microsoft Corporation

Introduction

.NET Framework 4 introduces new data structures that are specifically designed to simplify thread-safe access to shared data and increase the performance and scalability of multi-threading applications. In this document we present an overview of the four new collection types: ConcurrentQueue(T), ConcurrentStack(T), ConcurrentBag(T), and ConcurrentDictionary(TKey, TValue). We also discuss BlockingCollection(T), which wraps a variety of data structures with blocking and bounding logic.

Our aim is three-fold:

1. to discuss the implementation details in the data structures that relate to performance, although these details are subject to change in future releases.
2. to provide performance measurements that compare the data structures to alternatives and to measure the scalability of certain scenarios for different numbers of threads/cores
3. to provide best-practice guidance that will help answer questions such as "when should I use the new thread-safe collections," and "what aspects of a scenario make a given type more scalable or better performing than others?"

Throughout the document, our best-practice guidance is called out in boxes like the following:

> ✓ *Use the new functionality in .NET Framework 4 to get the most out of your multi-core machines*

First, we explain the performance measurements and the scenarios used for the majority of our analyses. We then discuss and analyze the performance of BlockingCollection(T), ConcurrentStack(T), ConcurrentQueue(T) , ConcurrentBag(T), and ConcurrentDictionary(T).

Our tests were run on a specific set of machines for which the configurations are described in the appendix. Our performance analyses were based primarily on the statistics of test completion time. We expect that the completion time will vary between runs and if the tests are run on different hardware. For this reason, the test results provided are to be used purely as a starting point of performance tuning for an application.

Performance Criteria

We measured the performance of the thread-safe collections to answer two questions:

1. For an algorithm using one of the thread-safe collections on a machine with at least *N* cores, how much faster does it execute when using *N* threads as compared to just using 1 thread? This measurement is called *scalability*.
2. How much faster is an algorithm - if it utilizes one of the new thread-safe collections, as opposed to an equivalent algorithm that doesn't use one of the new types? This measurement is called *speedup*.

The scalability investigations measured how the performance of a thread-safe collection varied as more cores were utilizedThe aim of the speedup investigations were quite different as they compare two ways of solving a problem using different approaches. In these investigations we typically used the same machine configuration and ensured the programs were unchanged except for the replacement of a thread-safe collection with an alternative that was equivalent in functionality. Each speedup experiment defined a single algorithm that required a thread-safe collection to operate correctly on multi-core machines. The experiments were run using one of the new thread-safe collections and, again, were compared against a simple implementation of the same data structure. The simple implementations are straightforward approaches built using types available prior to .NET Framework 4 – in most cases, they are a synchronization wrapper around a non-thread-safe collection. For example, the following SynchronizedDictionary(TKey, TValue) was compared to ConcurrentDictionary(TKey, TValue):

```
public class SynchronizedDictionary<TKey, TValue> : IDictionary<TKey, TValue>
{
    private Dictionary<TKey, TValue> m_Dictionary = new Dictionary<TKey, TValue>();
    private object _sync = new object();
    public void Add(TKey key, TValue value)
    {
        lock (_sync)
        {
            m_Dictionary.Add(key, value);
        }
    }
    // and so on for the other operations..
}
```

Producer-Consumer Scenarios Used for Speedup Comparisons

For our experiments, we created scenarios that we feel represent common usage. The most common scenarios that apply to the new collections are variations of the producer-consumer pattern (Stephen Toub, 2009). In particular, ConcurrentStack(T), ConcurrentQueue(T), and ConcurrentBag(T) are often appropriate for use as a buffer between producers and consumers. BlockingCollection(T) simply supports the blocking and bounding requirements of some producer-consumer scenarios.

In particular, we used two producer-consumer scenarios: a *pure* scenario and a *mixed* scenario.

In the pure scenario, we created *N* threads on an *N*-core machine where *N/2* threads only produced data and *N/2* threads only consumed data. This basic scenario appears whenever generation of work items is logically separate to the processing of work items. For simplicity, in this scenario we assume equal number of producers and

consumers and that they process each item with same speed. However in real world it is common that producers and consumers are not balanced.

The main loop for a producer thread typically looks like:

```
while (producingcondition())
{
    var item = DoDummyWork(workload); //simulates work to create the item
    collection.Add(item);
}
```

The main loop for a consumer thread typically looks like :

```
while (consumingcondition())
{
    TItem item;
    //TryTake returns true if item is removed successfully by the current
    //thread; false otherwise
    if (collection.TryTake(out item))
    {
        DoDummyWork(workload); //simulates work to process the item
    }
    else
    {
        //spin, wait, or do some other operations.
    }
}
```

The mixed scenario represents situations where threads both produce and consume data. Consider a generic tree-traversal algorithm over a family tree where each node is defined as:

```
public class Person
{
    public string name;
    public int age;
    public List<Person> children;
}
```

To traverse the tree, we can make use of any IProducerConsumerCollection(T) to act as a holding collection for subsequent nodes that must be traversed. The following code is a multi-threaded Traverse() method.

```
private static void Traverse(Person root,
    IProducerConsumerCollection<Person> collection)
{
    collection.TryAdd(root);
    Task[] tasks = new Task[dop];  //typically dop=n for n-core machine
    int activeThreadsNumber = 0;
    for (int i = 0; i < tasks.Length; i++)
    {
        tasks[i] = Task.Factory.StartNew(() =>
        {
            bool lastTimeMarkedFinished = false;
            Interlocked.Increment(ref activeThreadsNumber);
```

```
                while (true)
                {
                    if (lastTimeMarkedFinished)
                    {
                        Interlocked.Increment(ref activeThreadsNumber);
                        lastTimeMarkedFinished = false;
                    }
                    Person parent = null;
                    if (collection.TryTake(out parent))
                    {
                        foreach (Person child in parent.Children)
                        {
                            collection.TryAdd(child);
                        }
                        DoDummyWork(workload); //processing per node
                    }
                    else
                    {
                        if (!lastTimeMarkedFinished)
                        {
                            Interlocked.Decrement(ref activeThreadsNumber);
                            lastTimeMarkedFinished = true;
                        }

                        if (activeThreadsNumber == 0) //all tasks finished
                        {
                            return;
                        }
                    }
                }
            });
        }
        Task.WaitAll(tasks);
    }
```

In this program, each thread acts as both a producer and a consumer, adding and taking items from the shared collection.

In both the pure and mixed scenarios we simulated the work required to produce and consume items. We did this via a simple DoDummyWork(int *k*) function that repeats a simple floating point calculations *k* times. The exact details are not important but we can assume that each loop of the dummy function corresponds to a handful of simple machine instructions. Since we were interested in measuring the costs associated with accessing the data structures, we have typically used very small work functions.

BlockingCollection(T)

BlockingCollection(T) provides blocking and bounding semantics for any IProducerConsumerCollection(T) type. In this way, BlockingCollection(T) can add and remove elements using any underlying policy implemented by the backing collection while providing the mechanism to block on attempts to remove from an empty store and to

block on attempts to add if a bound is specified. Prior to .NET Framework 4, this would most likely have been constructed using a Monitor with a collection type like Queue(T), as the following NaiveBlockingQueue(T):

```
class NaiveBlockingQueue<T>
{
    Queue<T> m_queue;
    object m_lockObj;

    public NaiveBlockingQueue()
    {
        m_queue = new Queue<T>();
        m_lockObj = new object();
    }
    public void Add(T item)
    {
        lock (m_lockObj)
        {
            m_queue.Enqueue(item);
            Monitor.Pulse(m_lockObj);
        }
    }

    public void Take(out T item)
    {
        lock (m_lockObj)
        {
            while (m_queue.Count == 0)
                Monitor.Wait(m_lockObj);
            item = m_queue.Dequeue();
        }
    }
}
```

This NaiveBlockingQueue(T) blocks Take operations if the underlying queue is empty, it does not support rich functionalities such as bounding for additions, waking readers once all production has finished, cancellation and pluggable underlying collections. BlockingCollection(T) supports all of these.

NaiveBlockingCollection(T) uses a simple Monitor while BlockingCollection(T) internally uses more complex synchronization methods. From the performance point of view, NaiveBlockingCollection(T) could perform better in scenarios where workload is zero or very small. But once more functionalities are added to NaiveBlockingCollection(T), the performance starts to degrade with more overhead of synchronization.

To compare the performance of BlockingCollection(T) and NaiveBlockingQueue(T), we ran a test in which one thread produces all the elements and all the other threads concurrently consume these elements. We ran this test using 2, 4 and 8 threads with various workloads. With zero workload, BlockingCollection(T) performed worse than NaiveBlockingQueue(T) as expected, but as workload increased BlockingCollection(T) started to outperform NaiveBlockingQueue(T). We found the tipping point of the workload was 500 FLOPs in our test configuration, and this could vary according to different hardware. Figure 1 shows the elapsed time of this test running on an 8-core machine, for a variety of thread counts.

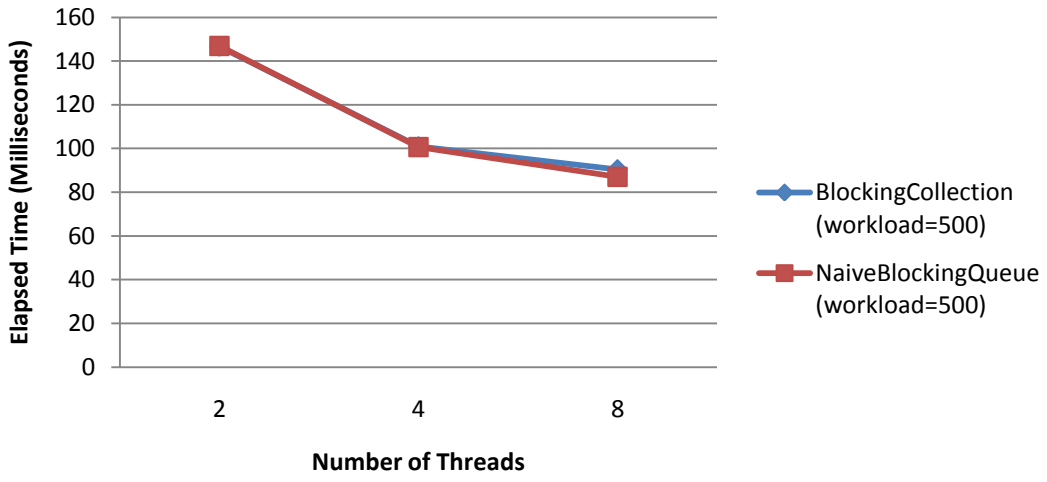

Figure 1: Comparison of scalability for BlockingCollection(T) and NaiveBlockingQueue(T) in the scenario with 1 producer and N-1 consumers and workload of 500FLOPs

The real power of the BlockingCollection(T) type, however, is that the rich functionalities it supports we mentioned. Given that BlockingCollection(T) performs similar with or better than BlockingQueue(T) and that it has much richer functionality, it is appropriate to use BlockingCollection(T) whenever blocking and bounding semantics are required.

✓ *When blocking and bounding semantics are required, BlockingCollection(T) provides both rich functionality and good performance.*

Other Considerations

Given a collection type, we may also be interested to get a count of items, to enumerate all the data, or to dump the data into another data structure. When using a BlockingCollection(T), it is useful to know the following performance characteristics.

The **Count** method relies on the synchronization mechanism of BlockingCollection(T), specifically the **CurrentCount** property of the SemaphoreSlim object it keeps internally. Thus Count is an O(1) operation, and it reflects the real count of the underlying collection, unless the underlying collection is modified outside the BlockingCollection(T), which is a bad practice that breaks the contract supported by BlockingCollection(T), and we advise against it.

BlockingCollection(T) also provides a **GetConsumingEnumerable**() method for the consumer to enumerate the collection, which, unlike normal enumerables, mutates the collection by repeatedly calling **Take**(). This means that

calling **MoveNext**() on this enumerator will block if the collection is empty and wait for the new items to be added into the collection. To safely stop enumerating, you can call **CompleteAdding**() or cancel the enumeration using the **GetConsumingEnumerable**(CancellationToken canellationToken) overload.

The **GetEnumerator**() method calls the underlying collection's **GetEnumerator**() method, so its performance depends on the implementation of the underlying collection. The **ToArray**() method of BlockingCollection(T) wraps calls to the corresponding methods of the underlying collection with minimal overhead, so its performance also depends on the underlying collection.

ConcurrentQueue(T)

ConcurrentQueue(T) is a data structure in .NET Framework 4 that provides thread-safe access to FIFO (First-In First-Out) ordered elements. Under the hood, ConcurrentQueue(T) is implemented using a list of small arrays and lock-free operations on the head and tail arrays, hence it is quite different than Queue(T) which is backed by an array and relies on the external use of monitors to provide synchronization. ConcurrentQueue(T) is certainly more safe and convenient than manual locking of a Queue(T) but some experimentation is required to determine the relative performance of the two schemes. In the remainder of this section, we will refer to a manually locked Queue(T) as a self-contained type called SynchronizedQueue(T).

Pure Producer-Consumer Scenario

The experiments we used for ConcurrentQueue(T) follow the producer-consumer patterns discussed earlier and we paid particular attention to the use of the simulated workload functions. The first experiment was a pure producer-consumer scenario where half of the threads were producers which simulated work by looping the simulated workload function then added an item to the queue; the other half were consumers which did the same simulation work but were instead removing items. The tests were run for various thread-counts and for differing workload sizes. We defined two workload sizes: the first is 0 FLOPS and the second is 500 FLOPS for both the producer loops and the consumer loops. These workloads are representative of workloads where contention would most likely be a dominating cost. For workloads that are significantly large, synchronization costs are potentially negligible. The exact values of elapsed times in milliseconds are not important since they vary by number of operations executed in the tests. Instead, we are interested in how elapsed time changes when a test runs on different numbers of threads, since it shows the scalability of this implementation.

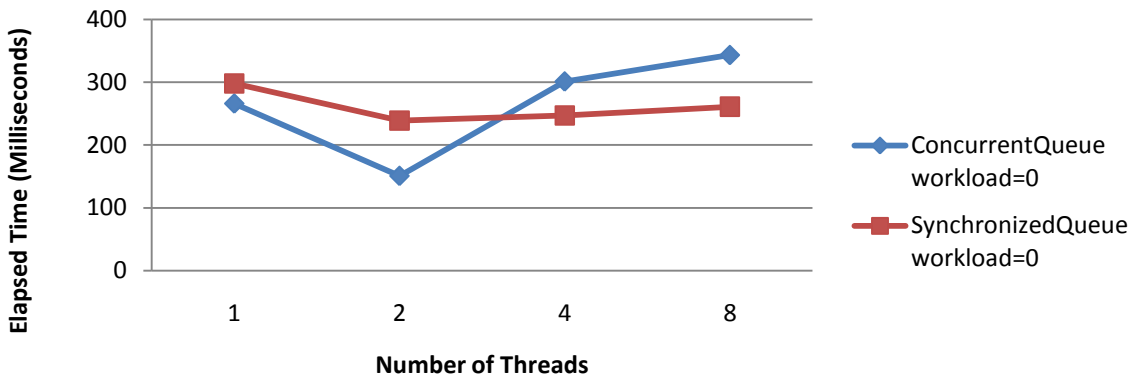

Figure 2: Comparison of scalability for ConcurrentQueue(T) and SynchronizedQueue(T) in a pure producer-consumer scenario with a zero-cost workload function.

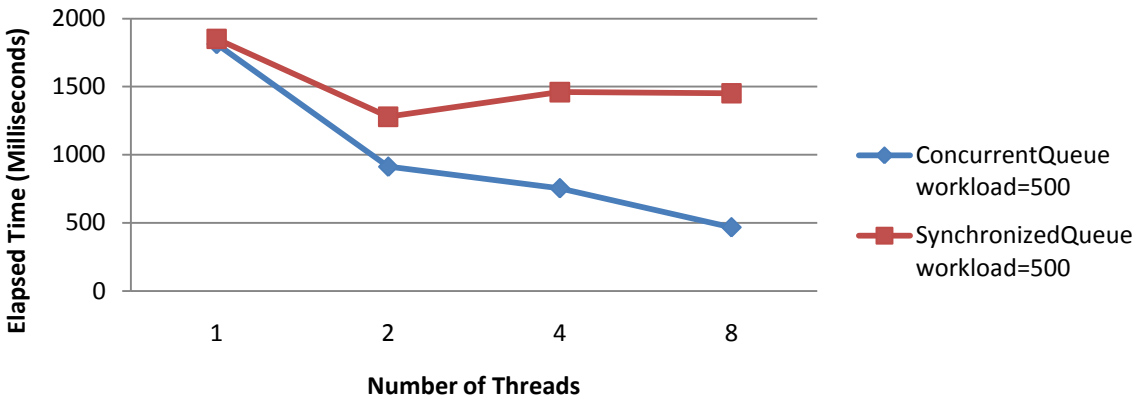

Figure 3: Comparison of scalability for ConcurrentQueue(T) and SynchronizedQueue(T) in a pure producer-consumer scenario with a 500 FLOPS workload function.

Figures 2 and 3 show the elapsed time for a pure producer-consumer scenario implemented using ConcurrentQueue(T) and SynchronizedQueue(T) with the two different workloads.

In Figure 2, we see that when the workload was very small, both ConcurrentQueue(T) and SynchronizedQueue(T) achieved their best performance when using exactly two threads and also that ConcurrentQueue(T) performed better for the two-thread case. The lack of scalability past two threads for both queues is expected as ConcurrentQueue(T) has only two access points (the head and the tail) and SynchronizedQueue(T) has only one access point because head and tail operations are serialized. So, at most, two threads can operate with little contention but more threads will necessarily suffer contention and synchronization overheads that will dominate in the absence of a significant workload function.

✓ *For scenarios with very light computation it is best to use ConcurrentQueue(T) on two threads: one pure producer, and the other pure consumer. Queues will not scale well beyond two threads for such scenarios.*

On the other hand, in Figure 3 we see that ConcurrentQueue(T) does scale beyond two threads for workloads of 500 FLOPS due to minimal synchronization overhead. SynchronizedQueue(T), however, does not scale for workloads of 500 FLOPS as its costs of synchronization are significantly higher and continue to be a significant factor. We found that 500 FLOPS is representative of the largest workload that shows a difference in scalability given the hardware our tests ran on. For larger workloads, the scalability of ConcurrentQueue(T) and Synchronized(T) do not differ greatly.

✓ *For scenarios involving moderate-size work functions (such as a few hundred FLOPS) , ConcurrentQueue(T) can provide substantially better scalability than SynchronizedQueue(T).*

If you have a small workload that falls in between 0 and 500 FLOPS, experimentation will best determine whether using more than two threads is beneficial.

Mixed Producer-Consumer Scenario

The second experiment for queues used the tree traversal scenario, in which each thread was both a producer and a consumer. Figure 4 and 5 show the results for this scenario.

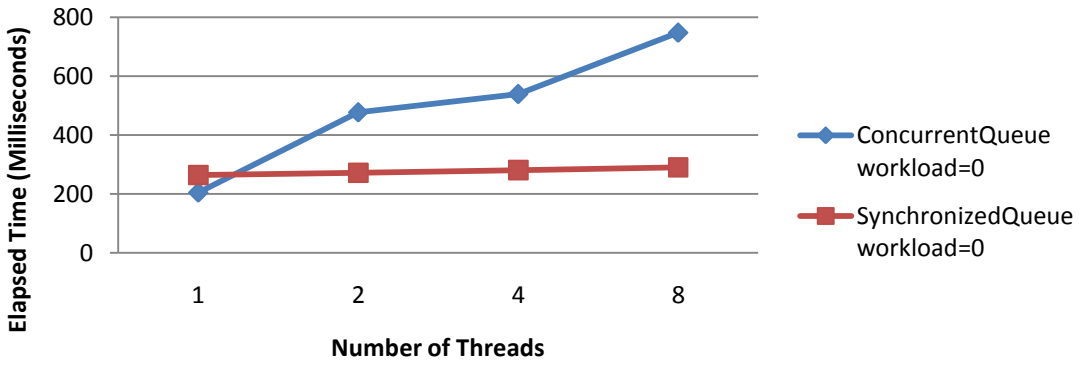

Figure 4: Comparison of scalability for ConcurrentQueue(T) and SynchronizedQueue(T) in a mixed producer-consumer scenario with a zero-cost workload function.

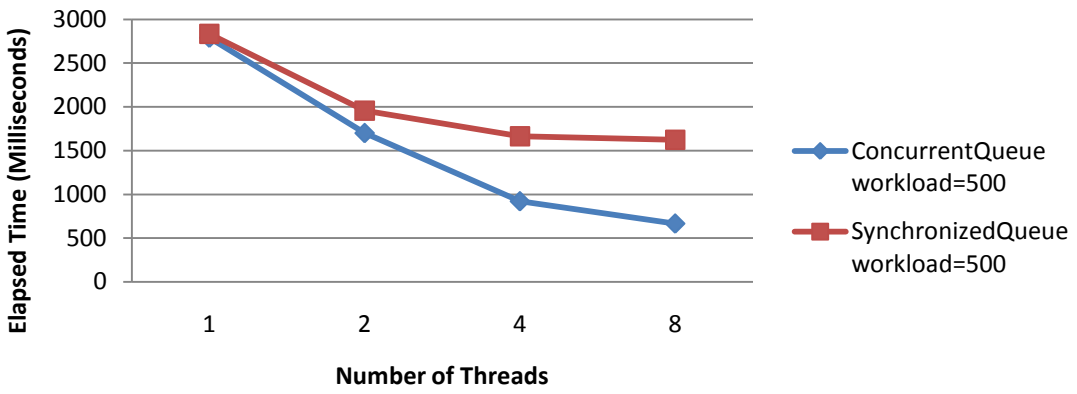

Figure 5: Comparison of scalability for ConcurrentQueue(T) and SynchronizedQueue(T) in a mixed producer-consumer scenario with a 500 FLOPs workload function.

From Figure 4, we see that neither data structure showed any scalability when the work function was very small.The loss of scalability for even the two-thread case was due precisely to this being a mixed producer-consumer scenario. The two threads were both performing operations on the head and the tail thus introducing contention that is not present in the pure scenario. ConcurrentQueue(T) performed worse than SynchronizedQueue(T) as we increase the number of threads in execution. This is due to the fact that the ConcurrentQueue(T) implementation uses compare-and-swap (CAS) primitives which rely on spinning to gain entry to critical resources (see the MSDN article on Interlocked Operations: http://msdn.microsoft.com/en-us/library/sbhbke0y.aspx). When contentious requests are as frequent as they are in this case, CAS primitives do not perform as well as locks, like those used in SynchronizedQueue(T).

However, as shown in Figure 5, when the work function is a few hundred FLOPS or larger, the mixed scenario showed scalability. In these situations, ConcurrentQueue(T) has lower overheads and thus shows significantly better performance that is amplified as the number of threads/cores increases.

> ✓ *For mixed producer-consumer scenarios, scalability is only available if the work function is a few hundred FLOPS or larger. For these scenarios, ConcurrentQueue(T) provides significantly better performance than SynchronizedQueue(T).*

Other Considerations

The **Count**, **ToArray**() and **GetEnumerator**() members take snapshots of the head and tail and, thus, the entire collection. Taking the snapshot is a lock-free operation and takes O(1) time on average, however, each of these members have their own additional costs.

Since the queue maintains an index for each item according to the order it is added to the queue, after the **Count** property takes the snapshot, it simply returns the result of substracting the head index from the tail index. Thus the **Count** property overall is O(1) on average.

After the **ToArray**() method takes the snapshot, it copies all the items into an array, thus it is overall an O(N) operation. **GetEnumerator**() delegates to **ToArray**() and returns the enumerator of the result array, thus it takes O(N) time to return, and provides an unchanging snapshot of items.

ConcurrentStack(T)

ConcurrentStack(T) is an implementation of the classic LIFO (Last-In First-Out) data structure that provides thread-safe access without the need for external synchronization. ConcurrentStack(T) is intended to be used in scenarios where multiple threads are managing a set of items and wish to process items in LIFO order. It is useful in scenarios for which new data should be processed in preference to processing older data, such as a multi-threaded depth-first search. Other examples arise in situations where there are penalties for not processing data on time. In such situations, the total penalties may be minimized by processing new items first, and allowing items that have already missed their schedule to be further delayed – if so, then a LIFO data structure for managing the items may be appropriate.

We compare the ConcurrentStack(T) to a simple implementation called SynchronizedStack(T) which is a thin wrapper around the non-thread-safe Stack(T) that uses a single monitor for synchronization.

Pure Producer-Consumer Scenario

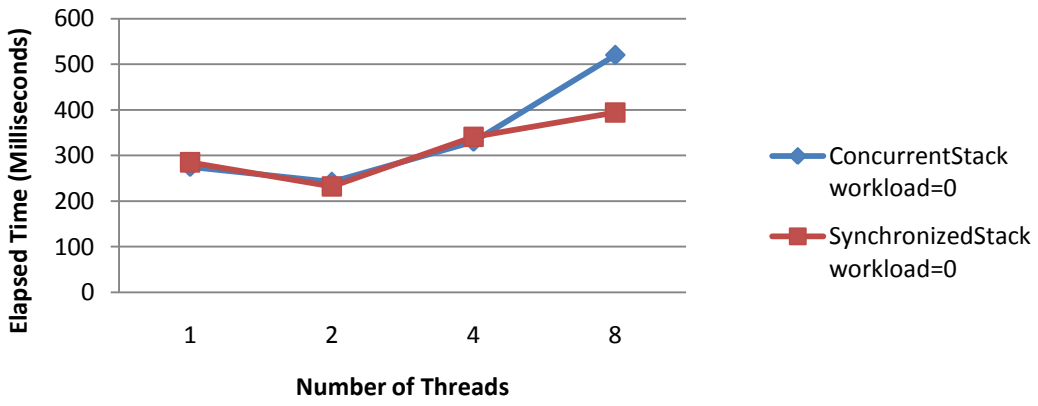

Figure 6: Comparison of scalability for ConcurrentStack(T) and SynchronizedStack(T) in a pure producer-consumer scenario with a zero-cost workload function.

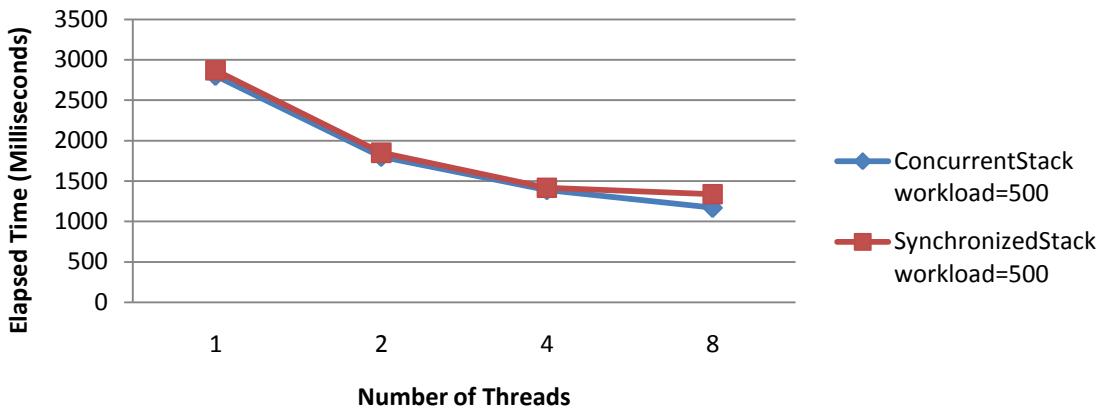

Figure 7: Comparison of scalability for ConcurrentStack(T) and SynchronizedStack(T) in a pure producer-consumer scenario with a 500 FLOPS workload function.

Figures 6 and 7 show the results for the a pure producer-consumer scenario implemented using ConcurrentStack(T) and SynchronizedStack(T). The tests used here are identical to that used for ConcurrentQueue(T).

From the results, we see that ConcurrentStack(T) has largely identical performance characteristics to SynchronizedStack(T). This is the result of both implementations having a single point of contention and the lack of opportunities for ConcurrentStack(T) to do anything particular to improve raw performance.

We also see in Figures 6 and 7 that a pure producer-consumer scenario involving stacks will only exhibit scaling if the workload is a few hundred FLOPS or larger. For smaller workloads, the scalability degrades until we see that no scalability is available when the workload is tiny.

✓ *For a pure producer-scenario scenario, ConcurrentStack(T) has essentially identical performance as a SynchronizedStack(T). Both show good scalability for work-functions that are a few hundred FLOPS or larger.*

Even though the performance characteristics are identical for this scenario, we recommend using the ConcurrentStack(T) due to it being simple and safe to use.

Mixed Producer-Consumer Scenario

Figures 8 and 9 show the results for the tree traversal scenario implemented using ConcurrentStack(T) and SynchronizedStack(T).

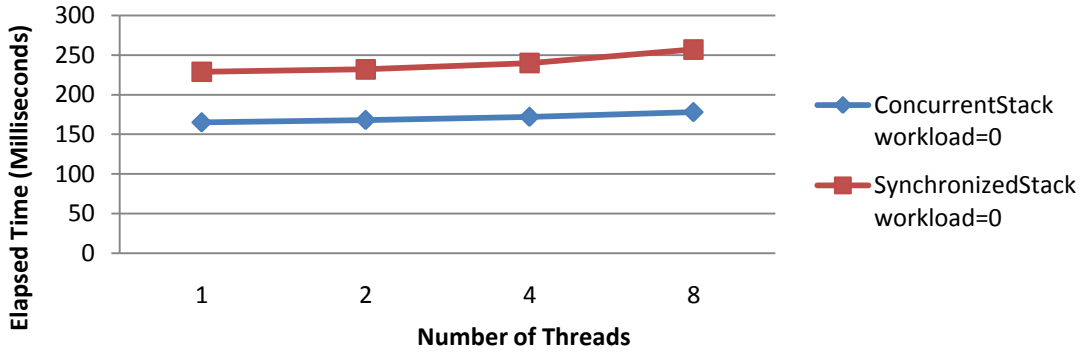

Figure 8: Comparison of scalability for ConcurrentStack(T) and SynchronizedStack(T) in a mixed producer-consumer scenario with a zero-cost workload function.

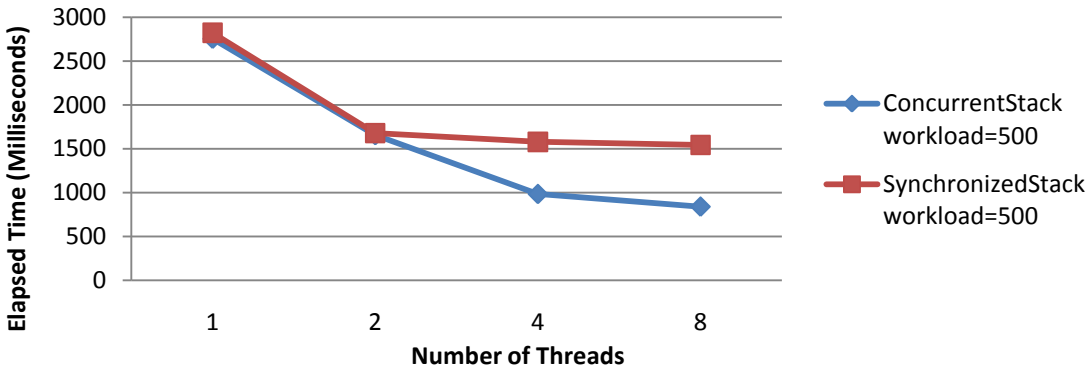

Figure 9: Comparison of scalability for ConcurrentStack(T) and SynchronizedStack(T) in a mixed producer-consumer scenario with a 500 FLOPS workload function.

In the tree traversal scenario, we actually see a divergence in the performance of the two implementations, and ConcurrentStack(T) has consistently better performance due to lower overheads when used in a mixed producer-consumer scenario.

> ✓ *For a mixed producer-scenario scenario, ConcurrentStack(T) has better performance than SynchronizedStack(T) and shows good better for work functions of a few hundred FLOPS or larger.*

Other Considerations

In some scenarios, you may have multiple items to add or remove at a time. For instance, when the LIFO order is preferred but not strictly required, a thread may be able to process N items at a time rather than processing them one-by-one. In this case, if we call **Push**() or **TryPop**() repetitively for N times, there is a synchronization cost for each operation. The **PushRange**() and **TryPopRange**() methods that ConcurrentStack(T) provide only use a single CAS operation to push or pop N items and thus significantly reduce the total synchronization cost in these scenarios.

> ✓ *For a scenario in which many items can be added or removed at a time to process, use PushRange() and TryPopRange() methods.*

It is worth mentioning that you can also implement PushRange() and TryPopRange() for SynchronizedStack(T), by taking the global lock to push or pop an array of items. It sometimes performs better than ConcurrentStack's PushRange() and TryPopRange(). This is because Stack(T), at the core of SynchronizedStack(T), is based on arrays, while ConcurrentStack(T) is implemented using a linked list which comes with memory allocation cost with each node. Nevertheless, we recommend to use ConcurrentStack(T) because it provides out of box API support for all scenarios, thread safety and decent overall performance.

The **Count**, **ToArray**() **and GetEnumerator**() members are all lock-free and begin by taking an immutable snapshot of the stack.

The **Count** method walks the stack to count how many items are present, and is thus an O(N) operation. Whenever possible, avoid accessing the **Count** property in a loop. For example, since the **IsEmpty** property takes only O(1) time, you should always use

```
while (!stack.IsEmpty)
{
    //do stuff
}
```

instead of

```
while (stack.Count > 0)
{
    //do stuff
}
```

The **ToArray**() and **GetEnumerator**() methods take a snapshot and then process the items in the snapshot, so they are both O(N) time operations.

ConcurrentBag(T)

ConcurrentBag(T) is a new type for .NET Framework 4 that doesn't have a direct counterpart in previous versions of .NET Framework. Items can be added and removed from a ConcurrentBag(T) as with ConcurrentQueue(T) and

ConcurrentStack(T) or any other IProducerConsumerCollection types, but the items are not maintained in a specific order. This lack of ordering is acceptable in situations where the only requirement is that all data produced is eventually consumed. Any scenario that can use a bag could alternatively use an ordered data structure such as a stack or a queue but the ordering rules demand restrictions and synchronization that can hamper scalability.

ConcurrentBag(T) is built on top of the new System.Threading.ThreadLocal(T) type such that each thread accessing the ConcurrentBag(T) has a private thread-local list of items. As a result, adding and taking items can often be performed locally by a thread, with very little synchronization overhead. However, a ConcurrentBag(T) must present a global view of all the data so, if a thread tries to take an item but finds its local list is empty, it will *steal* an item from another thread if other threads own items. Since ConcurrentBag(T) has very low overheads when each thread both adds and removes items, we can immediately see that the ConcurrentBag(T) should be an excellent collection type for mixed producer-consumer scenarios if ordering isn't a concern.

The graph traversal scenario is thus an ideal scenario for the ConcurrentBag(T) if the specific traversal ordering is not important. When the tree is balanced, there is a high probability that a thread that produces a node will also consume that node, so a signifcant percentage of TryTake() operations will be inexpensive removal operations from the thread's lock list, as opposed to costly steal operations from other thread lists. The process starts by adding the root node to the ConcurrentBag(T) on the main thread then spinning up producer-consumer threads. One of the threads will steal the root node and produce child nodes to search. From here, the other threads will race to steal nodes and then commence searching in their own sub-trees. Once the process warms up, the threads should largely operate in isolation with little need to synchronize until we start to run out of nodes to traverse.

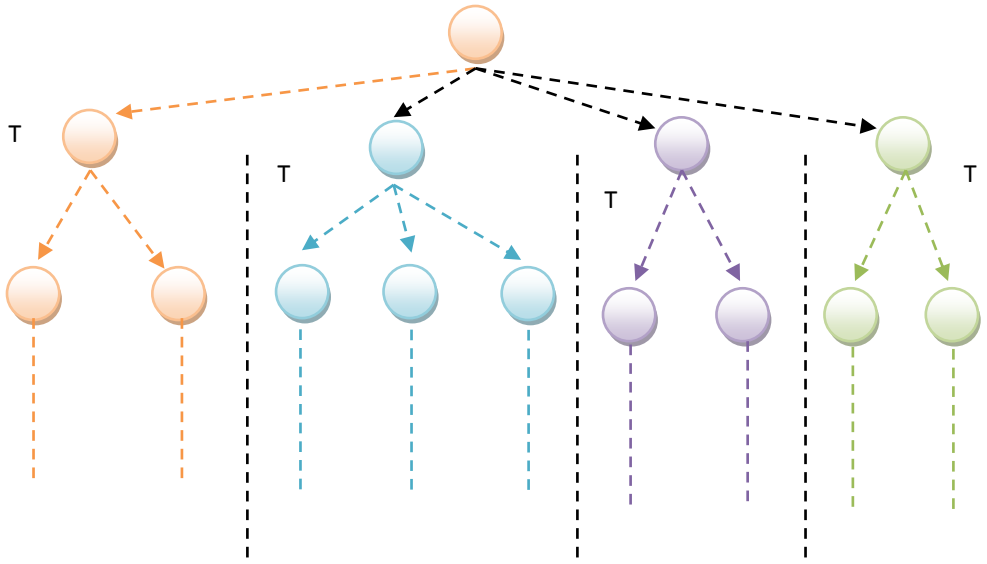

Figure 10: Visualization of graph-traversal using multiple-threads and a ConcurrentBag(T).

Figure 10 shows how a graph is traversed using a ConcurrentBag(T) and four threads. The black arrows show the nodes stolen by the threads and the node colors represent the thread used to traverse them.

The primary scenario we used to test ConcurrentBag(T) is an unordered graph-traversal where the work functions are string comparison (with a maximum of 10 characters in each string). The inner loop contains the following:

```
if (bag.TryTake(out node))
{
    for (int i = 0; i < node.Children.Count; i++)
    {
        bag.Add(node.Children[i]);
    }
    ProcessNode(node); //e.g. a short string comparison
}
```

For comparison types, the main contenders are thread-safe ordered collections such as thread-safe queue, and thread-safe stack. We could use simple implementations of the thread-safe ordered collections, but we chose to test against the new collections: ConcurrentQueue(T) and ConcurrentStack(T). Figure 11 shows the result of the tree-search scenario for different tree sizes.

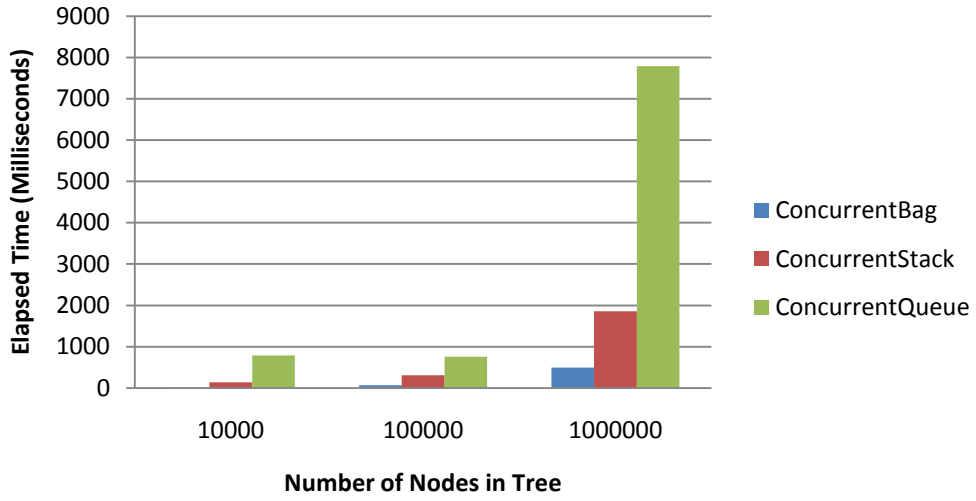

Figure 11: Comparison of ConcurrentBag(T), ConcurrentStack(T), and ConcurrentQueue(T)'s performance in a mixed producer-consumer scenario for various tree-sizes.

As expected, ConcurrentBag(T) dramatically outperformed other collections for this scenario and we can expect the results to generalize to other mixed producer-consumer scenarios.

> ✓ *For mixed producer-consumer scenarios that do not require item ordering, ConcurrentBag(T) can be dramatically more efficient than ConcurrentStack(T) , ConcurrentQueue(T), and other synchronized collections.*

To measure the scalability of ConcurrentBag(T), we ran the same scenario for a tree size of 100,000 nodes and varied only the number of threads involved in the search.

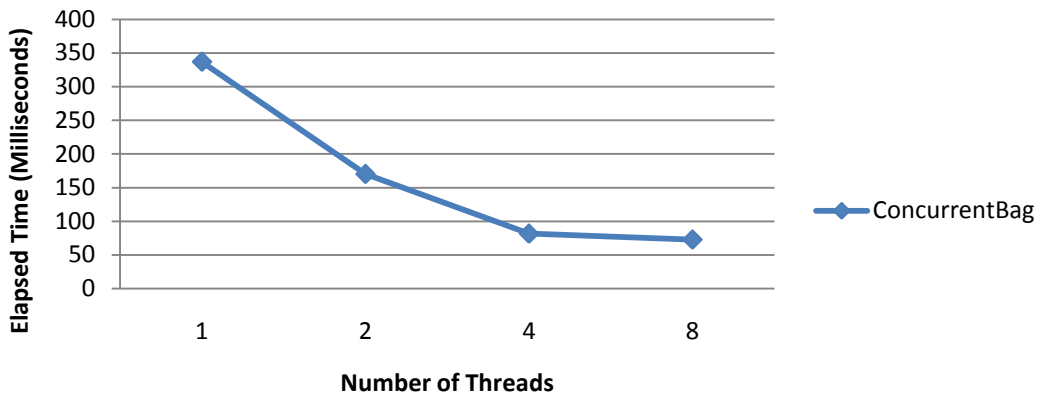

Figure 12: Scalability of ConcurrentBag(T) in a mixed producer-consumer scenario.

Figure 12 demonstrates that the scalability of ConcurrentBag(T) is excellent even when the work functions are very small. As noted previously, for scenarios that involve larger work functions, we can expect the scalability to be even closer to linear.

✓ *ConcurrentBag(T) shows excellent scalability for mixed producer-consumer scenarios.*

Although ConcurrentBag(T) has excellent performance for a mixed producer-consumer scenario, it will not have the same behavior for pure producer-consumer scenarios as all the consumers will have to repeatedly perform stealing operations and this will incur significant overheads and synchronization costs.

× *ConcurrentBag(T) may not be appropriate for pure producer-consumer scenarios.*

Other Considerations

ConcurrentBag(T) is a bit heavyweight from the memory prespective, the reason being that ConcurrentBag(T) is not disposable but it internally consists of disposable ThreadLocal(T) objects. These ThreadLocal(T) objects, even when they are not used anymore, cannot be disposed until the ConcurrentBag(T) object is collected by GC. **IsEmpty**, **Count, ToArray**() and **GetEnumerator**() lock the entire data structure so that they can provide a snapshot view of the whole bag. As such, these operations are inherently expensive and they cause concurrent **Add**() and **Take**() operations to block. Note that by the time **ToArray**() or **GetEnumerator**() returns, the global lock is already released and so the original collection may have already been modified.

ConcurrentDictionary(TKey,TValue)

The ConcurrentDictionary(TKey,TValue) type provides a thread-safe implementation of a strongly-typed dictionary. Prior to .NET Framework 4, the simple way to achieve thread-safe access to a strongly-typed dictionary structure was to use a lock to protect all accesses to a regular Dictionary(TKey,TValue). When using a locked Dictionary, the dictionary object itself can be used as the lock, so to safely update an element we would typically use the following:

```
void UpdateElement(Dictionary<int, int> dict, int key, int newValue)
{
    lock (dict)
    {
        dict[key] = updatedValue;
    }
}
```

When reading from this data structure, we must also take the lock as concurrent updaters may be making structural changes that make searching in the data structure impossible. Hence:

```
void GetElement(Dictionary<int, int> dict, int key)
{
    lock (dict)
    {
        return dict[key];
    }
}
```

Using a common lock effectively serializes all accesses to the data structure even if the bulk of the accesses are simple reads.

The ConcurrentDictionary(TKey,TValue) type provides a thread-safe dictionary which does not rely on a common lock. Rather, ConcurrentDictionary(TKey, TValue) internally manages a set of locks to provide safe concurrent updates and uses a lock-free algorithm to permit reads that do not take any locks at all.

The ConcurrentDictionary(TKey,TValue) is applicable to any problem involving concurrent access to a dictionary where updates are possible. However, for read-only access to a dictionary such as a lookup table with fixed data, a simple Dictionary(TKey,Value) has lower overheads than ConcurrentDictionary(TKey,TValue).

> ✓ *If you require only concurrent reads with no updates, a regular Dictionary(TKey,TValue) or a ReadOnlyDictionary(TKey,TValue) is appropriate, even in multi-threaded scenarios.*

It should also be noted that the Hashtable datastructure from .NET Framework 1.1 is intrinsically thread-safe for multiple non-enumerating readers and a single writer, but not safe for that multiple writers or enumerating readers. For certain situations, Hashtable is a reasonable baseline for comparison to ConcurrentDictionary(T), but we do not consider this further due to the more complex rules and because Hashtable is not intrisically a strongly-typed datastructure.

In the following sections, we will examine the performance of a ConcurrentDictionary(int,int) type in scenarios that involve various combinations of concurrent reads and writes. A dictionary is not typically used for producer-

consumer scenarios but rather in applications with lookup-tables and caches or when constructing groups of data with identical keys. As such, we will consider a different set of scenarios than those used previously.

An (Almost) Read-only Dictionary

Some problems call for a thread-safe dictionary that is mostly read-only but is occasionally updated. For example, consider a mapping of NetworkID to Status which changes when interruptions or repairs occur:

CentralNet	→	OK
ExternalNet	→	FAULT
LocalDeviceNet	→	OK

Because of the occasional updates, a multi-threaded application may only access this dictionary via completely thread-safe operations. The ConcurrentDictionary(TKey, TValue) has an opportunity to scale well due to its lock-free reads.

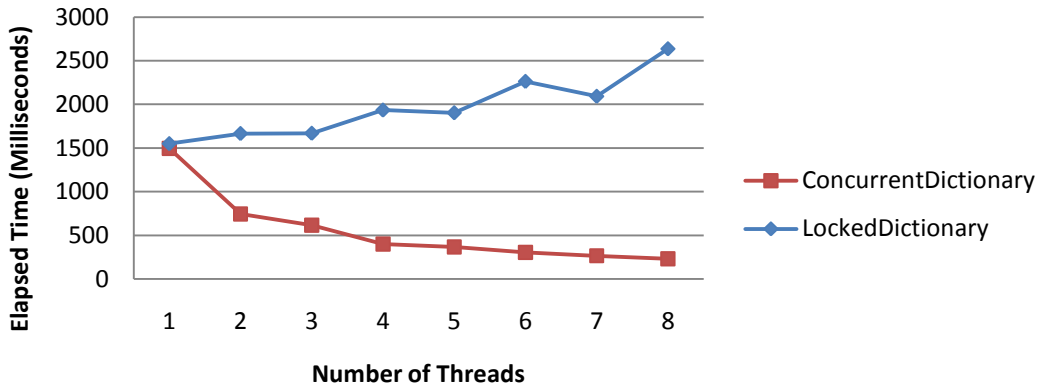

Figure 13: Comparison of scalability for ConcurrentDictionary(TKey,TValue) and SynchronizedStack(TKey,TValue) in a read-heavy producer-consumer scenario.

Figure 13 shows scalability for a scenario where a total of *M* reads were made in the absence of additions, deletions, or in-place updates. We can see that the ConcurrentDictionary(TKey, TValue) can service multiple threads performing concurrent reads and still scale well. On the other hand, the performance of a locked Dictionary(TKey, TValue) degrades as more threads participate as contention on the shared lock becomes an increasingly great cost.

✓ *For read-heavy scenarios that require a thread-safe dictionary, the ConcurrentDictionary(TKey,TValue) is the best choice.*

Frequent Updates

A variety of scenarios involve frequent adding and updating of values in a shared dictionary structure. For example, a dictionary might be used to accumulate item counts as data is extracted from a source. A simple thread-safe approach using a locked Dictionary(Int32,Int32) is:

```
void ExtractDataAndUpdateCount(Input input, Dictionary<int, int> dict)
{
    int data;
    ExtractDataItem(input, dict, out data);
    lock (dict)
    {
        if (!dict.ContainsKey(data))
            dict[data] = 1;
        else
            dict[data]++;
    }
}
```

If we look to use a ConcurrentDictionary(Int32,Int32), we need an approach that provides atomic updates without taking a common lock. One approach is to use a CAS loop that repeatedly reads an element and calls **Try Add**() if the element does not exist or **TryUpdate**() until it successfully updates without experiencing contention. Fortunately, ConcurrentDictionary(TKey,TValue) provides an **AddOrUpdate**() method which takes care of the details of performing an atomic update. **AddOrUpdate**() takes delegate parameters so that it can re-evaluate the updated value whenever write contention occurs. The corresponding code for ConcurrentDictionary(Int32,Int32) is thus:

```
void ExtractDataAndUpdateCount(Input input, ConcurrentDictionary<int, int> cd)
{
    int data;
    ExtractDataItem(input, dict, out data);
    cd.AddOrUpdate(data, k => 1, (k, v) => v + 1);
}
```

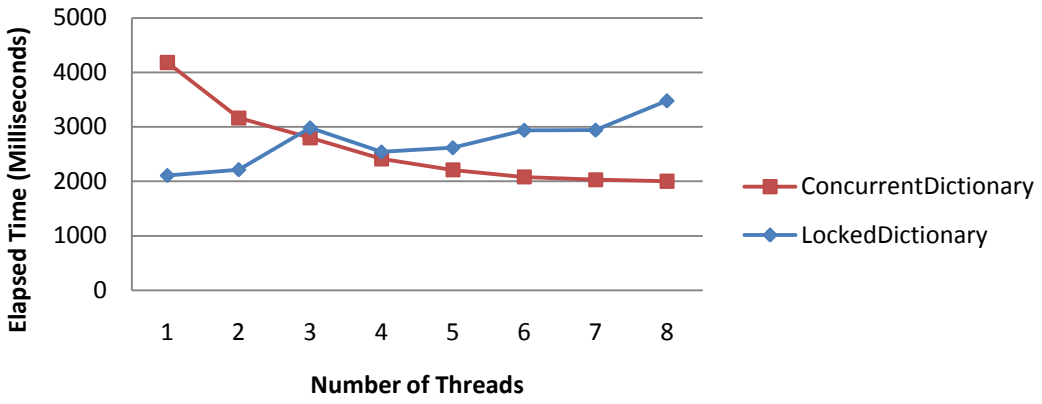

Figure 14: Comparison of scalability for ConcurrentDictionary(TKey,TValue) and SynchronizedStack(TKey,TValue) in an update-heavy producer-consumer scenario.

Figure 14 shows a comparison of performance for continuous atomic update operations. We assume that the ExtractDataItem() method is essentially free, and that there are many different key values so that we are not always updating the exact same items.

For frequent updates, LockedDictionary(TKey,TValue) shows a performance profile that is very similar to its read profile, which is expected given that both situations require taking a common lock and thus serialize all work.

The ConcurrentDictionary(TKey, TValue) data indicates that for sequential scenarios (nCores=1), the **AddOrUpdate**() operations are on the order of 0.5x the speed of a simple locked Dictionary(TKey, TValue). However, as the number of participating cores increases, the use of multiple internal locks within ConcurrentDictionary(TKey, TValue) permits some level of scalability. In particular, the update performance increases by up to a factor of 2 but it is limited by contention on the shared locks and cache-invalidation costs.

✓ *In a multi-threaded scenario that requires frequent updates to a shared dictionary, ConcurrentDictionary(TKey,TValue) can provide modest benefits.*

Since the scalability for frequent writes is not ideal, always looks for opportunities to rework an update-heavy algorithm such that the workers can accumulate data independently with a merge operation to combine them at the end of processing.

A scenario that entails all writes to a shared dictionary is the worst-case performance scenario for ConcurrentDictionary(TKey,TValue). A more realistic scenario may involve some significant time spent in the ExtractDataItem() method or other per-item processing. As more time is spent on local per-thread computation, scalability will naturally increase as contention on the dictionary will cease to be the primary cost. This applies equally to both LockedDictionary(TKey,TValue) and ConcurrentDictionary(TKey, TValue).

Concurrent Reading and Updating

We can also consider situations where some amount of reading and writing takes place. Recall that a simple locked Dictionary(TKey,TValue) must take locks for both reads and writes but that a ConcurrentDictionary(TKey,TValue) only requires locks for writes and that it manages multiple internal locks to provide some level of write-scalability.

The ConcurrentDictionary(TKey,TValue) is clearly the best choice whenever the level of concurrency is high, as it has better performance for both reads and writes. However, even in a dual-core scenario, we may find that ConcurrentDictionary(TKey,TValue) is the best choice if there is a significant proportion of reads.

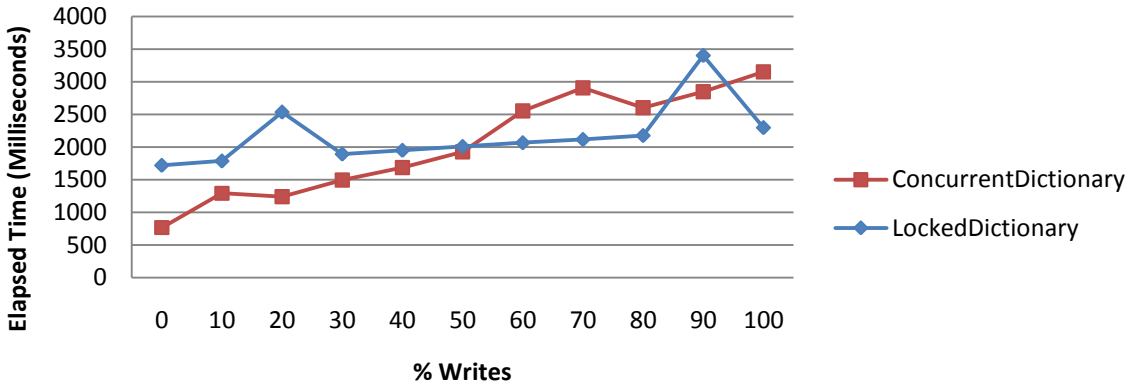

Figure 15: Comparison of performance for ConcurrentDictionary(TKey,TValue) and SynchronizedStack(TKey,TValue) for various read/update ratios.

> ✓ *If your scenario includes a significant proportion of reads to writes, ConcurrentDictionary(TKey, TValue) offers performance gains for any number of cores/threads.*

Other Considerations

ConcurrentDictionary(TKey,TValue)'s **Count**, **Keys**, **Values**, and **ToArray()** completely lock the data structure in order to provide an accurate snapshot. This serializes all of these calls and interferes with add and update performance. Hence, these methods and properties should be used sparingly.

The **GetEnumerator()** method provides an enumerator that can walk the Key/Value pairs that are stored in the dictionary. **GetEnumerator**() doesn't take any locks but it nonetheless guarantees that the enumerator is safe for use even in the face of concurrent updates. This is great for performance but, because no snapshot is taken, the enumerator may provide data that is a mixture of items present when **GetEnumerator**() was called and some or all of the subsequent updates that may have been made. If you require a an enumerable snapshot of the

ConcurrentDictionary(TKey, TValue), either arrange for all updates to pause before enumeration or use the **ToArray**() method to capture the data whilst all the internal locks are held.

ConcurrentDictionary(TKey,TValue) update operations are internally protected by fine-grain locks whose granularity can be tuned by specifying the concurrency level in the constructor:

```
public ConcurrentDictionary(int concurrencyLevel, int capacity)
```

If a concurrencyLevel is not specified, the default is four times the number of processors. Increasing the concurrency level increases the granularity of the locks which generally decreases contention for concurrent updates. As a result, specifying a concurrencyLevel higher than the default may improve performance for frequent-update scenarios. The downside is that all the operations that lock the whole dictionary may become significantly more expensive.

> ✓ *The default concurrency level is appropriate for most cases. For scenarios that include significant concurrent updates, you may wish to experiment with the concurreny level to acheive maximum performance.*

Details of the Experimental Framework

All tests were run on identical 8-core machines with the following hardware configurations:

- Intel ® Xeon® CPU E5345 @2. 3GHz, 2 sockets x 4-core.
- 8GB RAM
- .NET Framework 4 Beta 2
- All tests were run on both Windows Vista Ultimate 32-bit and Windows Vista Ultimate 64-bit

Each test was executed multiple times in a environment that reduced unnecessary external noise and we computed the mean elapsed time as the standard measure. Our results had standard deviations of less than 10% of the mean. We used several approaches to reduce noise in order to obtain stable results:

- We stopped all unnecessary background services and applications and turned off network access and peripherals where possible.
- We selected test parameters such that the each individual test iteration took at least 100 milliseconds. This reduced most noise.
- To reduce the impact of garbage collection, the tests kept memory use to a minimum by using simple types and small data volumes, and we forced garbage collections between runs.
- All timing measurements excluded the time required to initialize the application, warm up the .NET Framework ThreadPool, and perform other house-keeping operations.

Due to the nature of our tests, they do not represent entire applications. Thus the data we presented should be considered only indicative of the performance of similar units of a program on similar hardware, and not entire systems or applications.

Finally, we note that experiments run on 32-bit and a 64-bit platforms may show significant variance in both speedup and scalability. There are many factors that can be the cause this variance and, in some cases, the differences favor one architecture over the other. If maximum performance is crucial and an application can run on either a 32-bit or a 64-bit platform then specific performance measurements are required to select between the two alternatives. We structured our experiments and presentation of results to be agnostic to the specific choice of platform.

Bibliography

Microsoft Corp. (2007, Nov). *The Manycore Shift.* Retrieved from Microsoft: http://www.microsoft.com/downloads/details.aspx?FamilyId=633F9F08-AAD9-46C4-8CAE-B204472838E1

Stephen Toub. (2009, Nov). *Patterns for Parallel Programming: Understanding and Applying Parallel Patterns with the .NET Framework 4.* Retrieved from http://www.microsoft.com/downloads/details.aspx?FamilyID=86b3d32b-ad26-4bb8-a3ae-c1637026c3ee&displaylang=en