Parallel Containers and Objects

Visual Studio 2015

The Parallel Patterns Library (PPL) includes several containers and objects that provide thread-safe access to their elements.

A *concurrent container* provides concurrency-safe access to the most important operations. The functionality of these containers resembles those that are provided by the Standard Template Library (STL). For example, the concurrency::concurrent_vector class resembles the std::vector class, except that the **concurrent_vector** class lets you append elements in parallel. Use concurrent containers when you have parallel code that requires both read and write access to the same container.

A *concurrent object* is shared concurrently among components. A process that computes the state of a concurrent object in parallel produces the same result as another process that computes the same state serially. The concurrency::combinable class is one example of a concurrent object type. The **combinable** class lets you perform computations in parallel, and then combine those computations into a final result. Use concurrent objects when you would otherwise use a synchronization mechanism, for example, a mutex, to synchronize access to a shared variable or resource.

Sections

This topic describes the following parallel containers and objects in detail.

Concurrent containers:

concurrent_vector Class

The concurrency::concurrent_vector class is a sequence container class that, just like the std::vector class, lets you randomly access its elements. The **concurrent_vector** class enables concurrency-safe append and element access operations. Append operations do not invalidate existing pointers or iterators. Iterator access and traversal operations are also concurrency-safe.

Differences Between concurrent_vector and vector

The **concurrent_vector** class closely resembles the **vector** class. The complexity of append, element access, and iterator access operations on a **concurrent_vector** object are the same as for a **vector** object. The following points illustrate where **concurrent_vector** differs from **vector**:

- Append, element access, iterator access, and iterator traversal operations on a **concurrent_vector** object are concurrency-safe.
- You can add elements only to the end of a **concurrent_vector** object. The **concurrent_vector** class does not provide the **insert** method.
- A **concurrent_vector** object does not use move semantics when you append to it.
- The **concurrent_vector** class does not provide the **erase** or **pop_back** methods. As with **vector**, use the clear method to remove all elements from a **concurrent_vector** object.
- The **concurrent_vector** class does not store its elements contiguously in memory. Therefore, you cannot use the **concurrent_vector** class in all the ways that you can use an array. For example, for a

variable named **v** of type **concurrent_vector**, the expression `&v[0]+2` produces undefined behavior.

- The **concurrent_vector** class defines the grow_by and grow_to_at_least methods. These methods resemble the resize method, except that they are concurrency-safe.
- A **concurrent_vector** object does not relocate its elements when you append to it or resize it. This enables existing pointers and iterators to remain valid during concurrent operations.
- The runtime does not define a specialized version of **concurrent_vector** for type **bool**.

Concurrency-Safe Operations

All methods that append to or increase the size of a **concurrent_vector** object, or access an element in a **concurrent_vector** object, are concurrency-safe. The exception to this rule is the **resize** method.

The following table shows the common **concurrent_vector** methods and operators that are concurrency-safe.

at	end	operator[]
begin	front	push_back
back	grow_by	rbegin
capacity	grow_to_at_least	rend
empty	max_size	size

Operations that the runtime provides for compatibility with the STL, for example, **reserve**, are not concurrency-safe. The following table shows the common methods and operators that are not concurrency-safe.

assign	reserve
clear	resize
operator=	shrink_to_fit

Operations that modify the value of existing elements are not concurrency-safe. Use a synchronization object such as a reader_writer_lock object to synchronize concurrent read and write operations to the same

data element. For more information about synchronization objects, see Synchronization Data Structures.

When you convert existing code that uses **vector** to use **concurrent_vector**, concurrent operations can cause the behavior of your application to change. For example, consider the following program that concurrently performs two tasks on a **concurrent_vector** object. The first task appends additional elements to a **concurrent_vector** object. The second task computes the sum of all elements in the same object.

C++

```
// parallel-vector-sum.cpp
// compile with: /EHsc
#include <ppl.h>
#include <concurrent_vector.h>
#include <iostream>

using namespace concurrency;
using namespace std;

int wmain()
{
   // Create a concurrent_vector object that contains a few
   // initial elements.
   concurrent_vector<int> v;
   v.push_back(2);
   v.push_back(3);
   v.push_back(4);

   // Perform two tasks in parallel.
   // The first task appends additional elements to the concurrent_vector object.
   // The second task computes the sum of all elements in the same object.

   parallel_invoke(
      [&v] {
         for(int i = 0; i < 10000; ++i)
         {
            v.push_back(i);
         }
      },
      [&v] {
         combinable<int> sums;
         for(auto i = begin(v); i != end(v); ++i)
         {
            sums.local() += *i;
         }
         wcout << L"sum = " << sums.combine(plus<int>()) << endl;
      }
   );
}
```

Although the **end** method is concurrency-safe, a concurrent call to the push_back method causes the value that is returned by **end** to change. The number of elements that the iterator traverses is indeterminate. Therefore, this program can produce a different result each time that you run it.

Exception Safety

If a growth or assignment operation throws an exception, the state of the **concurrent_vector** object becomes invalid. The behavior of a **concurrent_vector** object that is in an invalid state is undefined unless stated otherwise. However, the destructor always frees the memory that the object allocates, even if the object is in an invalid state.

The data type of the vector elements, *_Ty*, must meet the following requirements. Otherwise, the behavior of the **concurrent_vector** class is undefined.

- The destructor must not throw.
- If the default or copy constructor throws, the destructor must not be declared by using the **virtual** keyword and it must work correctly with zero-initialized memory.

[Top]

concurrent_queue Class

The concurrency::concurrent_queue class, just like the std::queue class, lets you access its front and back elements. The **concurrent_queue** class enables concurrency-safe enqueue and dequeue operations. The **concurrent_queue** class also provides iterator support that is not concurrency-safe.

Differences Between concurrent_queue and queue

The **concurrent_queue** class closely resembles the **queue** class. The following points illustrate where **concurrent_queue** differs from **queue**:

- Enqueue and dequeue operations on a **concurrent_queue** object are concurrency-safe.
- The **concurrent_queue** class provides iterator support that is not concurrency-safe.
- The **concurrent_queue** class does not provide the **front** or **pop** methods. The **concurrent_queue** class replaces these methods by defining the try_pop method.
- The **concurrent_queue** class does not provide the **back** method. Therefore, you cannot reference the end of the queue.

- The **concurrent_queue** class provides the unsafe_size method instead of the **size** method. The **unsafe_size** method is not concurrency-safe.

Concurrency-Safe Operations

All methods that enqueue to or dequeue from a **concurrent_queue** object are concurrency-safe.

The following table shows the common **concurrent_queue** methods and operators that are concurrency-safe.

empty	push
get_allocator	try_pop

Although the **empty** method is concurrency-safe, a concurrent operation may cause the queue to grow or shrink before the **empty** method returns.

The following table shows the common methods and operators that are not concurrency-safe.

clear	unsafe_end
unsafe_begin	unsafe_size

Iterator Support

The **concurrent_queue** provides iterators that are not concurrency-safe. We recommend that you use these iterators for debugging only.

A **concurrent_queue** iterator traverses elements in the forward direction only. The following table shows the operators that each iterator supports.

Operator	Description
operator++	Advances to next item in the queue. This operator is overloaded to provide both pre-increment and post-increment semantics.
operator*	Retrieves a reference to the current item.

operator->	Retrieves a pointer to the current item.

[Top]

concurrent_unordered_map Class

The HYPERLINK "file:///C:\\Users\\thompet\\AppData\\Local\\Temp\\DxEditor\\DduePreview\\Default\\798d7037-df37-4310-858b-6f590bbf6ebf\\HTM\\html\\a217b4ac-af2b-4d41-94eb-09a75ee28622" concurrency::concurrent_unordered_map class is an associative container class that, just like the std::unordered_map class, controls a varying-length sequence of elements of type std::pair<const Key, Ty>. Think of an unordered map as a dictionary that you can add a key and value pair to or look up a value by key. This class is useful when you have multiple threads or tasks that have to concurrently access a shared container, insert into it, or update it.

The following example shows the basic structure for using **concurrent_unordered_map**. This example inserts character keys in the range ['a', 'i']. Because the order of operations is undetermined, the final value for each key is also undetermined. However, it is safe to perform the insertions in parallel.

C++

```
// unordered-map-structure.cpp
// compile with: /EHsc
#include <ppl.h>
#include <concurrent_unordered_map.h>
#include <iostream>

using namespace concurrency;
using namespace std;

int wmain()
{
    //
    // Insert a number of items into the map in parallel.

    concurrent_unordered_map<char, int> map;

    parallel_for(0, 1000, [&map](int i) {
        char key = 'a' + (i%9); // Geneate a key in the range [a,i].
        int value = i;          // Set the value to i.
        map.insert(make_pair(key, value));
    });

    // Print the elements in the map.
```

```
    for_each(begin(map), end(map), [](const pair<char, int>& pr) {
        wcout << L"[" << pr.first << L", " << pr.second << L"] ";
    });
}
/* Sample output:
    [e, 751] [i, 755] [a, 756] [c, 758] [g, 753] [f, 752] [b, 757] [d, 750] [h, 754]
*/
```

For an example that uses **concurrent_unordered_map** to perform a map and reduce operation in parallel, see How to: Perform Map and Reduce Operations in Parallel.

Differences Between concurrent_unordered_map and unordered_map

The **concurrent_unordered_map** class closely resembles the **unordered_map** class. The following points illustrate where **concurrent_unordered_map** differs from **unordered_map**:

- The **erase**, **bucket**, **bucket_count**, and **bucket_size** methods are named **unsafe_erase**, **unsafe_bucket**, **unsafe_bucket_count**, and **unsafe_bucket_size**, respectively. The **unsafe_** naming convention indicates that these methods are not concurrency-safe. For more information about concurrency safety, see Concurrency-Safe Operations.
- Insert operations do not invalidate existing pointers or iterators, nor do they change the order of items that already exist in the map. Insert and traverse operations can occur concurrently.
- **concurrent_unordered_map** supports forward iteration only.
- Insertion does not invalidate or update the iterators that are returned by **equal_range**. Insertion can append unequal items to the end of the range. The begin iterator points to an equal item.

To help avoid deadlock, no method of **concurrent_unordered_map** holds a lock when it calls the memory allocator, hash functions, or other user-defined code. Also, you must ensure that the hash function always evaluates equal keys to the same value. The best hash functions distribute keys uniformly across the hash code space.

Concurrency-Safe Operations

The **concurrent_unordered_map** class enables concurrency-safe insert and element-access operations. Insert operations do not invalidate existing pointers or iterators. Iterator access and traversal operations are also concurrency-safe. The following table shows the commonly used **concurrent_unordered_map** methods and operators that are concurrency-safe.

at	**count**	**find**	key_eq

begin	empty	get_allocator	max_size
cbegin	end	hash_function	operator[]
cend	equal_range	insert	size

Although the **count** method can be called safely from concurrently running threads, different threads can receive different results if a new value is simultaneously inserted into the container.

The following table shows the commonly used methods and operators that are not concurrency-safe.

clear	max_load_factor	rehash
load_factor	operator=	swap

In addition to these methods, any method that begins with **unsafe_** is also not concurrency-safe.

[Top]

concurrent_unordered_multimap Class

The concurrency::concurrent_unordered_multimap class closely resembles the **concurrent_unordered_map** class except that it allows for multiple values to map to the same key. It also differs from **concurrent_unordered_map** in the following ways:

- The concurrent_unordered_multimap::insert method returns an iterator instead of **std::pair<iterator, bool>**.
- The **concurrent_unordered_multimap** class does not provide **operator[]** nor the **at** method.

The following example shows the basic structure for using **concurrent_unordered_multimap**. This example inserts character keys in the range ['a', 'i']. **concurrent_unordered_multimap** enables a key to have multiple values.

C++

```
// unordered-multimap-structure.cpp
// compile with: /EHsc
#include <ppl.h>
#include <concurrent_unordered_map.h>
```

```
#include <iostream>

using namespace concurrency;
using namespace std;

int wmain()
{
    //
    // Insert a number of items into the map in parallel.

    concurrent_unordered_multimap<char, int> map;

    parallel_for(0, 10, [&map](int i) {
        char key = 'a' + (i%9); // Geneate a key in the range [a,i].
        int value = i;          // Set the value to i.
        map.insert(make_pair(key, value));
    });

    // Print the elements in the map.
    for_each(begin(map), end(map), [](const pair<char, int>& pr) {
        wcout << L"[" << pr.first << L", " << pr.second << L"] ";
    });
}
/* Sample output:
    [e, 4] [i, 8] [a, 9] [a, 0] [c, 2] [g, 6] [f, 5] [b, 1] [d, 3] [h, 7]
*/
```

[Top]

concurrent_unordered_set Class

The concurrency::concurrent_unordered_set class closely resembles the **concurrent_unordered_map** class except that it manages values instead of key and value pairs. The **concurrent_unordered_set** class does not provide **operator[]** nor the **at** method.

The following example shows the basic structure for using **concurrent_unordered_set**. This example inserts character values in the range ['a', 'i']. It is safe to perform the insertions in parallel.

C++

```
// unordered-set-structure.cpp
// compile with: /EHsc
#include <ppl.h>
#include <concurrent_unordered_set.h>
#include <iostream>

using namespace concurrency;
```

```
using namespace std;

int wmain()
{
    //
    // Insert a number of items into the set in parallel.

    concurrent_unordered_set<char> set;

    parallel_for(0, 10000, [&set](int i) {
        set.insert('a' + (i%9)); // Geneate a value in the range [a,i].
    });

    // Print the elements in the set.
    for_each(begin(set), end(set), [](char c) {
        wcout << L"[" << c << L"] ";
    });
}
/* Sample output:
    [e] [i] [a] [c] [g] [f] [b] [d] [h]
*/
```

[Top]

concurrent_unordered_multiset Class

The concurrency::concurrent_unordered_multiset class closely resembles the **concurrent_unordered_set** class except that it allows for duplicate values. It also differs from **concurrent_unordered_set** in the following ways:

- The concurrent_unordered_multiset::insert method returns an iterator instead of **std::pair<iterator, bool>**.
- The **concurrent_unordered_multiset** class does not provide **operator[]** nor the **at** method.

The following example shows the basic structure for using **concurrent_unordered_multiset**. This example inserts character values in the range ['a', 'i']. **concurrent_unordered_multiset** enables a value to occur multiple times.

C++

```
// unordered-set-structure.cpp
// compile with: /EHsc
#include <ppl.h>
#include <concurrent_unordered_set.h>
#include <iostream>
```

```
using namespace concurrency;
using namespace std;

int wmain()
{
    //
    // Insert a number of items into the set in parallel.

    concurrent_unordered_multiset<char> set;

    parallel_for(0, 40, [&set](int i) {
        set.insert('a' + (i%9)); // Geneate a value in the range [a,i].
    });

    // Print the elements in the set.
    for_each(begin(set), end(set), [](char c) {
        wcout << L"[" << c << L"] ";
    });
}
/* Sample output:
    [e] [e] [e] [e] [i] [i] [i] [i] [a] [a] [a] [a] [a] [c] [c] [c] [c] [c] [g] [g]
    [g] [g] [f] [f] [f] [f] [b] [b] [b] [b] [b] [d] [d] [d] [d] [d] [h] [h] [h] [h]
*/
```

[Top]

combinable Class

The concurrency::combinable class provides reusable, thread-local storage that lets you perform fine-grained computations and then merge those computations into a final result. You can think of a **combinable** object as a reduction variable.

The **combinable** class is useful when you have a resource that is shared among several threads or tasks. The **combinable** class helps you eliminate shared state by providing access to shared resources in a lock-free manner. Therefore, this class provides an alternative to using a synchronization mechanism, for example, a mutex, to synchronize access to shared data from multiple threads.

Methods and Features

The following table shows some of the important methods of the **combinable** class. For more information about all the **combinable** class methods, see combinable Class.

Method	Description

local	Retrieves a reference to the local variable that is associated with the current thread context.
clear	Removes all thread-local variables from the **combinable** object.
combine combine_each	Uses the provided combine function to generate a final value from the set of all thread-local computations.

The **combinable** class is a template class that is parameterized on the final merged result. If you call the default constructor, the *_Ty* template parameter type must have a default constructor and a copy constructor. If the *_Ty* template parameter type does not have a default constructor, call the overloaded version of the constructor that takes an initialization function as its parameter.

You can store additional data in a **combinable** object after you call the combine or combine_each methods. You can also call the **combine** and **combine_each** methods multiple times. If no local value in a **combinable** object changes, the **combine** and **combine_each** methods produce the same result every time that they are called.

Examples

For examples about how to use the **combinable** class, see the following topics:

- How to: Use combinable to Improve Performance
- How to: Use combinable to Combine Sets

[Top]

Related Topics

How to: Use Parallel Containers to Increase Efficiency
Shows how to use parallel containers to efficiently store and access data in parallel.

How to: Use combinable to Improve Performance
Shows how to use the **combinable** class to eliminate shared state, and thereby improve performance.

How to: Use combinable to Combine Sets
Shows how to use a **combine** function to merge thread-local sets of data.

Parallel Patterns Library (PPL)
: Describes the PPL, which provides an imperative programming model that promotes scalability and ease-of-use for developing concurrent applications.

Reference

concurrent_vector Class

concurrent_queue Class

concurrent_unordered_map Class

concurrent_unordered_multimap Class

concurrent_unordered_set Class

concurrent_unordered_multiset Class

combinable Class