**병렬 컨테이너와 오브젝트**

PPL에서는 thread-safe 하게 사용할 수 있는 몇 가지의 병렬 컨테이너와 오브젝트를 제공합니다. **먼저 병렬 컨테이너는 기존 STL 컨테이너와 매칭되며 주요 메서드를 thread-safe 하게 사용할 수 있습니다.** 그 예로 concurrency::concurrent\_vector 클래스는 std::vector 클래스와 비슷하며 컨테이너 원소를 여러 스레드에서 동시에 추가할 수 있습니다. 하지만 앞서 주요 메서드라고 언급하였듯이 모든 메서드가 thread-safe 한 것은 아니라는 것은 알아두셔야 합니다. 그리고 앞으로 설명하게 될 병렬 컨테이너들은 모두 STL 컨테이너와 매칭되므로 차이점 위주로 설명을 드리도록 하겠습니다.

**병렬 오브젝트는 여러 스레드가 공유하여 동시에 사용할 수 있는 객체를 의미합니다.** PPL 에서는 concurrency::combinable 클래스가 병렬 오브젝트에 속하며 별도의 동기화 객체 없이 스레드별로 계산 작업을 수행하고 최종 결과를 병합하는데 사용합니다. 그럼 먼저 concurrency::combinable 클래스부터 살펴보도록 하죠.

**combinable**

일반적으로 병렬 작업에서 스레드간에 객체를 공유할 때 뮤텍스와 같은 객체를 이용하여 동기화를 구현합니다. 하지만 이렇게 객체에 접근할 때마다 동기화 하는것은 성능에 악영향을 미치고 하나의 스레드를 이용한 방식보다 오히려 성능이 떨어질 수도 있습니다. **concurrency::combinable 클래스는 병렬 작업을 처리할 때 동기화 객체를 사용하지 않고 각각의 스레드에서 thread-local storage에 계산 작업을 저장하고 작업이 완료되면 각각의 결과값을 하나로 병합하는 역할을 합니다.**이는 여러 스레드 또는 task 간에 객체를 공유하여 사용할 경우에 유용하게 사용할 수 있습니다.

thread-local storage에 대한 개념을 아직 모르시는 분들은 검색을 통해 먼저 이 개념을 숙지하고나서 강좌를 읽으시는 것을 추천해드립니다. 그럼 combinable 클래스의 예제로 컨테이너에 담긴 정수값이 소수인지 판단하고 모든 소수 값의 합을 구하는 코드를 작성해보겠습니다. 이것은 이미 6장의 parallel\_transform과 parallel\_reduce 예제에서도 나왔던 것인데 이번에는 combinable 클래스를 이용하여 작성해보도록 하죠.

|  |
| --- |
| #include <windows.h> #include <ppl.h> #include <array> #include <numeric> #include <iostream>  using namespace concurrency; using namespace std;  // 전달받은 함수를 호출하고 함수 실행 소요 시간을 반환합니다. template <class Function> \_\_int64 time\_call(Function&& f) {     \_\_int64 begin = GetTickCount();     f();     return GetTickCount() - begin; }  // 전달받은 값이 소수인지 아닌지 판단하여 반환합니다. bool is\_prime(int n) {     if (n < 2)         return false;     for (int i = 2; i < n; ++i)     {         if ((n % i) == 0)             return false;     }     return true; }  int wmain() {     // 20만개의 정수값을 가지는 배열을 생성합니다.     array<int, 200000> a;      // 각 원소의 값을 인덱스 값과 같게(a[i] = i) 설정합니다.     iota(begin(a), end(a), 0);      int prime\_sum = 0;     \_\_int64 elapsed;      // 배열의 원소 중 모든 소수의 합을 계산합니다.     elapsed = time\_call([&] {         **for\_each(begin(a), end(a), [&](int i) {             prime\_sum += (is\_prime(i) ? i : 0);         });**     });     wcout << prime\_sum << endl;     wcout << L"serial time: " << elapsed << L" ms" << endl << endl;      // 같은 작업을 병렬 알고리즘과 combinable 클래스를 사용하여 실행합니다.     elapsed = time\_call([&] {        **combinable<int> sum;         parallel\_for\_each(begin(a), end(a), [&](int i) {             sum.local() += (is\_prime(i) ? i : 0);         });         prime\_sum = sum.combine(plus<int>());**     });     wcout << prime\_sum << endl;     wcout << L"parallel time: " << elapsed << L" ms" << endl << endl; }    // 실행결과  1709600813 serial time: 6797 ms  1709600813 parallel time: 1735 ms |

**combinable 클래스의 local 메서드는 계산 작업을 수행하는 각 스레드의 thread-local storage 변수를 반환합니다. 스레드별로 독립된 변수이기 때문에 동기화작업이 필요없죠**. parallel\_for\_each 로 수행한 병렬 작업이 완료되면 combinable 클래스의 combine 메서드를 사용하여 각 스레드의 thread-local storage 변수를 더하여 모든 소수의 합을 구할 수 있습니다. 그리고 combine 메서드의 동작은 다음과 같이 combine\_each 메서드를 통해서도 구현할 수 있습니다.

|  |
| --- |
| prime\_sum = 0; sum.combine\_each([&](int local){     prime\_sum += local; }); |

위의 예제처럼 parallel\_for\_each가 처리하는 작업이 시간이 많이 소요되는 작업(is\_prime)일 경우에는 병렬로 처리하였을때 성능이 향상되지만 시간이 적게 소요되는 단순한 연산일 경우에는 오히려 병렬로 처리하였을때 성능이 떨어질 수도 있다는걸 유념하시기 바랍니다.

**concurrent\_vector**

concurrency::concurrent\_vector 클래스는 std::vector 클래스에 대응하는 병렬 컨테이너입니다.**std::vector 클래스와 마찬가지로 랜덤 엑세스가 가능하며 원소의 추가 및 엑세스 작업은 thread-safe 함을 보장합니다.** 그리고 원소 추가 작업이 iterator를 무효화시키지 않기 때문에 iterator 엑세스및 traversal 작업 역시 thread-safe 합니다. 그럼 지금부터는 std::vector 클래스와의 차이점을 살펴보도록 하죠.

* 원소의 추가, 엑세스, iterator 엑세스, traversal 작업은 thread-safe 하다.
* 원소는 맨 뒤로만 추가할 수 있으며 insert 메서드는 제공되지 않는다.
* 원소의 추가 작업에 move semantics 가 적용되지 않는다.
* erase 메서드와 pop\_back 메서드가 제공되지 않으며, 원소를 삭제하려면 clear 메서드를 사용하여 모든 원소를 삭제해야 한다.
* 원소를 연속된 메모리 공간에 저장하지 않기 때문에 std::vector 처럼 &v[0] + 2 와 같은 표현식을 사용할 수 없다.
* resize 메서드와 비슷한 역할을 하면서 thread-safe 한 메서드인 grow\_by와 grow\_to\_at\_least 메서드를 제공한다.

차이점을 나열하다보니 제약 사항이 꽤 많네요. 아무래도 thread-safe 하게 구현하면서도 성능 저하는 최소화 하려다보니 이리 저리 제약 사항이 많이 생기게 된 것 같습니다. 이번에는 concurrent\_vector 클래스 메서드 중 thread-safe 한 메서드와 그렇지 않은 메서드를 구분하여 정리해보겠습니다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **메서드** | **thread\_safe** | **메서드** | **thread\_safe** | **메서드** | **thread\_safe** |
| at | O | end | O | operator[] | O |
| begin | O | front | O | push\_back | O |
| back | O | grow\_by | O | rbegin | O |
| capacity | O | grow\_to\_at\_least | O | rend | O |
| empty | O | max\_size | O | size | O |
| assign | X | reserve | X | clear | X |
| resize | X | operator= | X | shrink\_to\_fit | X |

위에서 언급된 thread-safe 하지 않은 메서드들은 여러 스레드에서 동시에 호출하면 예기치 않은 오류가 발생할 수 있으니 주의하시기 바랍니다. 그럼 마지막으로 concurrent\_vector 클래스를 사용하는 예제를 살펴보도록 하죠.

|  |
| --- |
| #include <ppl.h> #include <concurrent\_vector.h> #include <iostream>  using namespace concurrency; using namespace std;  int wmain() {     // concurrent\_vector 객체를 생성하고 몇 개의 원소를 추가합니다.     concurrent\_vector<int> v;     v.push\_back(2);     v.push\_back(3);     v.push\_back(4);       // 두 개의 작업을 병렬로 처리합니다.     // 첫 번째 작업은 concurrent\_vector에 원소를 추가하고     // 두 번째 작업은 concurrent\_vector를 traversal 하여 모든 원소의 합을 구합니다.     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;         }     ); } |

두 개의 작업을 병렬로 처리하는데 한 쪽에서는 컨테이너에 계속 원소를 추가하고 한 쪽에서는 컨테이너의 모든 원소들을 traversal 하면서 그 합을 구합니다. 이 예제가 출력하는 결과는 무엇일까요? push\_back 메서드와 end 메서드는 thread-safe 하기 때문에 서로 다른 스레드에서 호출되더라도 안전합니다. 그리고 첫 번째 작업의 push\_back 으로 인하여 두 번째 작업의 end 메서드의 반환값은 호출할 때 마다 바뀔 수 있기 때문에 이 예제의 출력 결과는 가늠할 수 없고 실행할 때마다 출력 결과가 달라질 수 있습니다.

**concurrent\_queue**

concurrency::concurrent\_queue 클래스는 std::queue 클래스에 대응하는 병렬 컨테이너입니다.**std::queue 클래스와 마찬가지로 맨 앞의 원소에 엑세스가 가능하며 enqueue 작업과 dequeue 작업은 thread-safe 함을 보장합니다.** 그리고 iterator를 제공하지만 이는 thread-safe 하지 않습니다. 이번엔 역시 std::queue 클래스와의 차이점을 살펴보도록 하죠.

* enqueue 작업과 dequeue 작업은 thread-safe 하다.
* iterator를 제공하지만 thread-safe 하지 않다.
* front 메서드와 pop 메서드를 제공하지 않고 dequeue 작업을 위해 thread-safe 한 try\_pop 메서드를 제공한다.
* back 메서드를 제공하지 않기 때문에 큐의 맨 뒤 원소를 참조할 수 없다.
* size 메서드 대신 unsafe\_size 메서드를 제공하며 이는 이름에서 알 수 있다시피 thread-safe 하지 않다.

concurrency::concurrent\_queue 클래스 역시 제약사항이 많이 있네요. 그리고 특이하게도 큐에서 iterator를 제공하지만 이는 thread-safe 하지 않기 때문에 디버깅할 때에만 사용하는 것을 권장하고 있습니다. 마지막으로 thread-safe 한 메서드와 그렇지 않은 메서드를 메서드를 정리해보도록 하겠습니다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **메서드** | **thread\_safe** | **메서드** | **thread\_safe** | **메서드** | **thread\_safe** |
| empty | O | push | O | get\_allocator | O |
| try\_pop | O | clear | X | unsafe\_end | X |
| unsafe\_begin | X | unsafe\_size | X | operator++ | X |
| operator\* | X | operator-> | X |  |  |

병렬 컨테이너와 오브젝트 강좌도 내용이 길어져서 두 장으로 나누어 진행해야겠네요. 다음 장에는 unordered\_map과 unordered\_set에 대응하는 병렬 컨테이너에 대하여 살펴보겠습니다.