**병렬 알고리즘**

PPL에서는 데이터 집합(배열, vector 등등..)을 기반으로 작업을 동시에 처리할 수 있는 알고리즘 함수들을 제공합니다. **이런 병렬 알고리즘 함수들은 기존에 STL에서 사용하던 알고리즘 함수와 사용 방법이 거의 비슷하다는 장점을 가지고 있습니다**. 이러한 장점 때문에 STL 알고리즘 함수를 사용해 본 경험이 있는 분들은 별 다른 어려움 없이 코드를 이해할 수 있을 뿐만 아니라 **기존에 STL 알고리즘을 이용하여 구현된 코드를 PPL의 병렬 알고리즘으로 쉽게 변환할 수 있습니다.** PPL의 병렬 알고리즘은 병렬 프로그래밍과 스레드에 대한 특별한 기반 지식 없이도 쉽게 사용할 수 있기 때문에 아마도 이번 강좌가 제 병렬 프로그래밍 강좌 중에서 가장 쓰임새 있는 강좌가 되지 않을까 싶네요.

**parallel\_for**

가장 먼저 소개해 드릴 함수는 parallel\_for 함수입니다. 이름에서 이미 예상할 수 있듯이 for 루프의 내용을 병렬로 실행해주는 함수이지요. **parallel\_for 함수는 루프 안에서 자원을 공유하지 않고 서로 독립적인 작업을 할 경우에 유용하게 사용될 수 있습니다.** parallel\_for 함수를 비롯해 이후에 설명하게될 대부분의 병렬 알고리즘 함수는 내부적으로 (지난 강좌에서 배운) structured\_task\_group 클래스를 이용하여 구현되어 있으며 work-stealing 알고리즘 등을 이용하여 최적화된 방법으로 작업을 분할하여 균형있게 처리합니다. 그러므로 사용자들은 몇 개의 스레드가 실행되고 어떻게 작업이 분배되는지 전혀 알 필요가 없습니다. 단지 작업이 '동시에' 처리될 수 있다는 점만 유념하여 코드를 구현해주면 됩니다.

parallel\_for 함수에는 몇 가지 오버로드 버전이 존재하는데 그 중 가장 대표적인 버전은 첫 번째 파라미터에 시작 인덱스를, 두 번째 파라미터에는 종료 인덱스를 전달하고 세 번째 파라미터에 작업 함수(함수형)를 전달하여 호출합니다. 또 다른 오버로드 버전은 시작 인덱스와 종료 인덱스 뒤로 증가될 값의 크기(Step)를 전달하고 마지막에 작업 함수를 전달하여 호출합니다.

기존의 for 루프를 병렬로 처리하기 위한 대체 수단으로 parallel\_for 함수를 이용할 수 있지만, for 루프와 parallel\_for 함수의 차이점을 유의하여 작성하여야 합니다. 그 차이점을 정리해보면 다음과 같습니다.

* parallel\_for 함수가 처리하는 작업의 실행 순서는 정해져 있지 않습니다.
* for 루프에서는 break를 통해 도중에 루프를 빠져나올 수 있지만 parallel\_for 함수는 모든 반복 작업이 끝난 이후에 종료됩니다.
* 시작 인덱스와 종료 인덱스는 반드시 정수형(양수, 음수)이어야 합니다.
* 루프의 진행은 순방향으로만 가능합니다. 이는 증가될 값의 크기(Step)가 음수값이 될 수 없음을 의미합니다. 만약 Step 값이 1보다 작으면 std::invalid\_argument 예외가 발생합니다.

parallel\_for 함수의 임의 중단은 task 그룹의 취소를 이용하여 구현할 수 있으며 이는 이후 Cancellation 강좌에서 자세히 다루도록 하겠습니다. 그럼 이제 기다리던 예제를 한 번 살펴 보도록 하죠.

|  |
| --- |
| // 병렬 알고리즘을 사용하려면 ppl.h 파일을 포함시켜야 합니다. **#include <ppl.h>** #include <array> #include <iostream>  using namespace concurrency; using namespace std;  int wmain() {     // 정수값을 담는 배열을 생성합니다.     array<int, 5> values1 = { 1, 2, 3, 4, 5 };     array<int, 5> values2 = { 2, 3, 4, 5, 6 };     array<int, 5> results;      // values1 배열의 값과 values2 배열의 값을 더하여 results 배열에 저장합니다.     **parallel\_for(0, 5, [&](int index)** {         results[index] = values1[index] + values2[index];     });      // results 배열의 값을 출력합니다.     for (int i = 0; i < 5; ++i)     {         wcout << results[i] << L"  ";     } }  // 실행결과  3  5  7  9  11 |

이 예제에서는 정수를 담고있는 두 배열에서 같은 인덱스에 해당하는 원소의 합을 또 다른 배열에 담아 출력하는 것을 parallel\_for 함수를 이용하여 구현하고 있습니다. 여기서는 parallel\_for 함수의 시작 인덱스를 0으로 설정하고, 종료 인덱스를 5로 설정하였는데 람다 함수의 index 변수에 전달되는 값이 0~5 값이 아닌 0~4 값이라는 것만 주의하시면 됩니다.

**parallel\_for 함수는 기존의 for 루프와 마찬가지로 중첩 루프도 구현이 가능합니다.**작업 함수에 람다 함수를 사용하면 기존 방식 그대로 직관적인 방법으로 중첩된 루프를 구현할 수 있죠. 그럼 이번엔 parallel\_for 함수를 중첩 루프로 구현한 예제를 하나 더 만들어보도록 하죠.

|  |
| --- |
| #include <ppl.h> #include <array> #include <iostream>  using namespace concurrency; using namespace std;  int wmain() {     // 구구단 테이블 생성합니다.     array<array<int, 9>, 8> multi\_table;      // parallel\_for 함수를 중첩하여 사용하여 구구단 테이블을 병렬로 채웁니다.     **parallel\_for(2, 10, [&](int i)** {         **parallel\_for(1, 10, [&](int j)** {             multi\_table[i - 2][j - 1] = i \* j;         });     });      // 구구단 테이블을 출력합니다.     for (auto it1 = multi\_table.begin(); it1 != multi\_table.end(); ++it1)     {         for (auto it2 = it1->begin(); it2 != it1->end(); ++it2)         {             wcout << \*it2 << L" ";         }         wcout << endl;     } }    // 실행결과  2  4  6  8  10  12  14  16  18  3  6  9  12  15  18  21  24  27 4  8  12  16  20  24   28  32  36  5  10  15  20  25  30  35  40  45 6  12  18  24  30  36  42  48  54 7  14  21  28  35  42  49  56  63 8  16  24  32  40  48  56  64  72  9  18  27  36  45  54  63  72  81 |

구구단 코드를 병렬 알고리즘을 이용해 만들어보니 또 감회가 새롭네요. ^^; 이 예제에서는 구구단 테이블의 값을 중첩된 parallel\_for 함수를 이용하여 채우고 있습니다. 어떤가요..기존 for 루프와 전혀 다르지 않죠? 이어서 설명할 parallel\_for\_each 함수도 중첩에 관한 내용은 parallel\_for 함수와 같으므로 그 부분에 대해서는 따로 설명드리지 않도록 하겠습니다.

**parallel\_for\_each**

**parallel\_for\_each 함수는 iterator 기반의 컨테이너와 함께 사용하며 컨테이너의 범위를 지정하여 지정된 범위 안의 데이터를 병렬로 처리하는 함수입니다.** parallel\_for\_each 함수의 동작 방식은 이미 살펴본 parallel\_for와 유사하며 내부적으로 같은 알고리즘을 사용하여 구현되어 있습니다. parallel\_for\_each 함수는 데이터를 병렬로 처리하기 때문에 실행 순서를 보장받을 수 없다는 것을 제외하면 STL의 알고리즘 함수인 for\_each 함수와 매우 유사합니다. 우리는 이미 for\_each 함수에 익숙하기 때문에 parallel\_for\_each 함수 역시 어렵지 않게 작성할 수 있죠. 그럼 바로 예제를 살펴보겠습니다.

|  |
| --- |
| #include <ppl.h> #include <array> #include <sstream> #include <iostream>  using namespace concurrency; using namespace std;  int wmain() {     // 정수값을 담는 배열을 생성합니다.     array<int, 5> values = { 1, 2, 3, 4, 5 };      // 배열에 담긴 값을 병렬로 출력합니다.     **parallel\_for\_each(begin(values), end(values), [](int value)** {         wstringstream ss;         ss << value << L' ';         wcout << ss.str();     }); }  // 실행결과  4  5  1  2  3 |

이번 예제는 MSDN 예제를 그대로 가져와 봤습니다. 코드를 봐도 딱히 설명할 것이 없는 초간단 예제네요. 병렬 프로그래밍이 이렇게 쉬워질 수 있다니... 참으로 고마운 일입니다. ^^ 작업의 실행 순서가 보장 되지 않기 때문에 배열 값의 출력 순서가 실행 할 때 마다 달라질 수 있다는 것만 유의하시면 될 것 같네요.

지금까지 parallel\_for 함수와 parallel\_for\_each 함수에 대하여 알아보았습니다. 두 함수는 PPL 중에서도 가장 쉬운 방법으로 병렬 프로그래밍을 구현할 수 있는 도구가 아닌가 생각이 됩니다.