**parallel\_invoke**

**parallel\_invoke 함수는 여러개의 작업을 병렬로 실행해주는 함수입니다.** 지난 task 그룹 강좌를 보신 분들이라면 parallel\_invoke 함수가 낯이 익으실 지도 모르겠네요. 그 당시에 structured\_task\_group 클래스를 설명하면서 간결하고 직관적인 코드 작성을 위해 structured\_task\_group 객체를 직접 생성하여 사용하는것 보다 parallel\_invoke 함수를 사용하는 것이 좋다고 설명드린 적이 있습니다. **parallel\_invoke 함수는 서로 독립적인 작업 몇 개를 동시에 실행하고자 할 때 유용하게 사용할 수 있습니다.**

parallel\_invoke 함수는 파라미터로 작업 함수(함수형)를 전달받으며 작업 함수는 최소 2개부터 최대 10개까지 전달할 수 있습니다. 결국 총 9개의 오버로드 버전을 제공한다는 얘기가 되겠네요. 그러므로 병렬로 처리할 작업의 개수가 10개 이하일 경우에는 parallel\_invoke 함수를 이용하여 구현할 수 있습니다. parallel\_invoke 함수는 파라미터로 전달 받은 모든 작업이 종료될 때까지 대기하며 작업의 순서는 보장되지 않습니다. 그럼 예제를 보면서 어떻게 사용하는지 알아보도록 하죠.

|  |
| --- |
| #include <ppl.h> #include <string> #include <iostream>  using namespace concurrency; using namespace std;  // 전달받은 값을 + 연산자로 더하여 반환합니다. template <typename T> T twice(const T& t) {     return t + t; }  int wmain() {     int n = 54;     double d = 5.6;     wstring s = L"Hello";      // 여러 타입의 값을 각각 더하는 함수를 동시에 실행합니다.     **parallel\_invoke(**         **[&n] { n = twice(n); },**         [&d] { d = twice(d); },         [&s] { s = twice(s); }     );      // 결과 값을 출력합니다.     wcout << n << L' ' << d << L' ' << s << endl; }  // 실행결과  108 11.2 HelloHello |

먼저 전달받은 값을 더하여 반환하는 템플릿 함수 twice가 정의되어 있구요, 서로 다른 타입의 변수를 사용하여 twice 함수를 호출하는 작업 함수 3개를 parallel\_invoke 함수에 전달하여 호출합니다. 만약 이와 같은 작업을 structured\_task\_group을 이용하여 구현했다면 structured\_task\_group 객체를 생성하고 각각의 작업에 해당하는 task\_handle 3개를 생성하여 run 시키고 마지막에 wait 하도록 구현하였겠죠. 하지만 위의 예제처럼 parallel\_invoke 함수를 이용하면 단 몇 줄의 코드로 구현이 가능하다는 것을 알 수 있습니다.

**parallel\_transform과 parallel\_reduce**

**parallel\_transform 함수는 STL 알고리즘 함수인 std::transform 함수의 병렬 버전으로 컨테이너에서 지정한 범위의 데이터를 변형하고 그 결과값을 자신의 컨테이너 혹은 또 다른 컨테이너에 담는 함수입니다.** parallel\_transform 함수는 이미지의 밝기를 조절하는 이미지 프로세스 처리,  벡터의 내적을 구하는 연산, 3D Ray-tracing 처리 등 병렬 프로세싱으로 인한 성능 향상을 기대할 수 있을만큼 큰 데이터를 처리할 때 유용하게 사용됩니다.

마찬가지로 parallel\_reduce 함수는 STL 알고리즘 함수인 std::accumulate 함수의 병렬 버전으로 컨테이너에서 지정한 범위의 데이터를 이용하여 하나의 결과값을 도출해내는 용도로 사용하는 함수입니다. parallel\_reduce 함수는 연속된 행렬의 곱셈 처리, 연속된 문자열 집합의 결합 또는 길이 계산 등에 활용될 수 있습니다. **그리고 보통의 경우 parallel\_for\_each와 concurrency::combinable 조합하여 사용한 코드를 parallel\_reduce 함수로 대체하여 구현이 가능합니다.**

parallel\_transform, parallel\_reduce 함수는 내부 처리 작업을 병렬로 실행한다는 점과 그에 따라 연산 순서가 정해져 있지 않다는 것을 제외하면 기존 STL의 transform, accumulate 함수와 동일하기 때문에 사용 방법에 대한 자세한 설명은 생략하도록 하겠습니다. 대신에 컨테이너에 담긴 정수값이 소수인지 판단하고 모든 소수 값의 합을 구하는 코드를 기존 STL 알고리즘 함수를 이용한 방법과 병렬 알고리즘을 이용한 방법 두 가지로 나누어 구현하고 각각의 처리 시간을 출력하는 예제를 만들어보도록 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;     \_\_int64 elapsed;      // 배열의 원소 중 모든 소수의 합을 계산합니다.     elapsed = time\_call([&] {         **transform(begin(a), end(a), begin(a), [](int i)** {             return is\_prime(i) ? i : 0;         });         **prime\_sum = accumulate(begin(a), end(a), 0);**     });     wcout << prime\_sum << endl;     wcout << L"serial time: " << elapsed << L" ms" << endl << endl;      // 배열 값을 다시 설정합니다.     iota(begin(a), end(a), 0);      // 같은 작업을 병렬 알고리즘을 사용하여 실행합니다.     elapsed = time\_call([&] {        **parallel\_transform(begin(a), end(a), begin(a), [](int i)** {             return is\_prime(i) ? i : 0;         });        **prime\_sum = parallel\_reduce(begin(a), end(a), 0);**     });     wcout << prime\_sum << endl;     wcout << L"parallel time: " << elapsed << L" ms" << endl << endl; }    // 실행결과  1709600813 serial time: 6797 ms  1709600813 parallel time: 1735 ms |

예제를 보시면 STL 알고리즘 함수인 transform, accumulate 함수와 병렬 알고리즘 함수인 parallel\_transform, parallel\_reduce 함수의 사용법은 완전히 동일하다는 것을 볼 수 있습니다. 이것은 STL 알고리즘을 이용하여 구현된 기존 코드를 단지 Ctrl+H(찾아 바꾸기) 한 번만으로도 병렬 알고리즘으로 변환할 수 있다는 이야기가 됩니다. 참 쉽죠? ^^ 그렇다면 이제 출력 결과를 보도록 하겠습니다. 저 결과는 제 PC 환경(Intel Core i7 CPU, 8GB RAM)에서 직접 실행해본 결과인데요, 병렬 알고리즘을 사용하여 구현했을 경우 무려 3.9배의 성능 향상이 일어났습니다.