**1. 작업량이 작은 루프는 병렬 처리하지 말것**

PPL을 이용한 병렬 처리는 리소스를 최대한 활용하여 작업을 빠르게 처리할 수 있다는 장점을 가지고 있지만 이에 따른 스케줄링 오버헤드도 발생합니다.**만약 병렬 루프 안에서 작은 양의 작업을 수행할 경우에는 스케줄링 오버헤드가 병렬 처리로 얻는 이득보다 커지는 상황이 발생할 수 있습니다.** 지금 설명하고 있는 '작업량이 작은 루프'는 루프 안에서 처리하는 내용의 크기를 얘기하는 것이지 루프의 반복 횟수를 얘기하는 것이 아니라는 것을 유념하시기 바랍니다. 그럼 이에 관한 예제를 살펴보도록 하죠.

|  |
| --- |
| #include <ppl.h> #include <iostream>  using namespace concurrency; using namespace std;  int wmain() {     // 같은 크기를 가지는 3개의 정수형 배열을 생성     const size\_t size = 100000;     int a[size], b[size], c[size];      // 배열 a와 b의 값을 초기화     for (size\_t i = 0; i < size; ++i)     {         a[i] = i;         b[i] = i \* 2;     }      // 배열 a와 b의 원소값의 합을 병렬 처리를 통해 배열 c에 저장     parallel\_for<size\_t>(0, size, [&a,&b,&c](size\_t i) {         **c[i] = a[i] + b[i];**     });      // TODO: 배열 c를 이용한 작업 수행 } |

위의 예제에서는 parallel\_for 루프 안에서 단지 배열 a와 b의 원소값을 더하여 배열 c에 넣어주는 작업만 처리하고 있습니다. 이렇게 병렬 루프에서 처리하는 작업량이 작을 경우에는 성능이 오히려 감소할 수 있습니다. **그러므로 이런 경우에는 병렬 루프 안에서 좀 더 많은 작업을 수행하도록 설계하거나 일반적인 for 루프로 변경함으로서 성능을 향상시킬 수 있습니다.**

**2. 병렬 루프 안에서 작업이 반복적으로 블러킹(blocking)되지 않게 할것**

일반적으로 병렬 루프를 사용하는 이유는 독립적인 작업을 동시에 실행하여 빠르게 처리하기 위함입니다. **하지만 병렬 루프 안에서 처리하는 작업이 자주 블러킹된다면 작업을 처리하는 스레드의 대기시간이 발생하게 되어 병렬 처리의 효율이 크게 떨어지게 되죠.** 게다가 Concurrency Runtime 에서는 협동적 블러킹(Cooperative Blocking) 개념을 구현하고있는데 작업이 이러한 방식으로 블러킹될 경우 다음과 같은 추가적인 문제점이 발생합니다.

Concurrency Runtime 에서는 Task가 블러킹되어 대기하고 있을 경우 스케줄러가 리소스 사용의 극대화를 위해 대기중인 작업을 새로 시작하게 되는데 이를 협동적 블러킹이라고 얘기하며, 만약 남아있는 스레드가 없을 경우에 새로운 스레드를 생성하여 실행합니다. **만약 작업이 가끔 한 번씩 블러킹되는 상황이라면 이러한 메카니즘이 성능 향상에 도움이 되지만 작업이 빈번하게 블러킹되는 상황에서는 단시간 안에 쓸데없이 많은 스레드가 생성되어 작업의 효율이 떨어지게됩니다.** 아래 예제에서는 병렬 루프 안에서 블러킹되지 않는 일반적인 작업을 수행할 경우와 협동적 블러킹을 메카니즘을 사용하는 작업을 수행할 경우 사용되는 스레드 개수의 차이를 보여줍니다.

|  |
| --- |
| #include <ppl.h> #include <agents.h> #include <iostream>  using namespace concurrency; using namespace std;  int wmain() {     combinable<int> dummy1, dummy2;     int threadCount1 = 0, threadCount2 = 0;      // 일반적인 병렬 루프     parallel\_for(0, 100000, [&dummy1](int i) {         // 스레드 개수 계산을 위한 dummy work         dummy1.local();          // 작업 수행...     });      // 병렬 루프가 작업 처리에 사용한 스레드 개수 계산     dummy1.combine\_each([&threadCount1](int num) {         ++threadCount1;     });      wcout << L"First loop thread count: " << threadCount1 << endl;      // 메시지 버퍼 생성     **overwrite\_buffer<int> buffer;**      // 병렬 루프 안에서 버퍼에 데이터 전송     parallel\_for(0, 100000, [&buffer, &dummy2](int i) {          // 스레드 개수 계산을 위한 dummy work         dummy2.local();          // send 함수는 동시에 여러 스레드에서 호출될 경우         // 협동적 블러킹 메카니즘에 의해 동기화됨         **send(buffer, i);**     });      // 병렬 루프가 작업 처리에 사용한 스레드 개수 계산     dummy2.combine\_each([&threadCount2](int num) {         ++threadCount2;     });      wcout << L"Second loop thread count: " << threadCount2 << endl; }  // 실행결과  First loop thread count: 9 Second loop thread count: 116 |

먼저 overwrite\_buffer 클래스는 메시지를 저장하는 버퍼를 가지고 있으며 마지막으로 저장된 메시지를 반환하는 역할을 하고, send 함수는 협동적 블러킹 메카니즘에 의해 동기화되어 버퍼에 메시지를 저장하는 역할을 합니다. 출력 결과를 살펴보면 첫번째 parallel\_for 루프에서는 9개의 스레드를 사용하여 작업을 수행한 반면, 두번째 parallel\_for 루프에서는 무려 116개의 스레드를 생성하여 작업을 수행한 것을 볼 수 있습니다. 따라서 위와 같이 블러킹 될 수 있는 작업은 별도의 스레드를 생성하여 일반적인 for 루프 안에서 처리하는 식으로 구현하는 것이 좋습니다.

**3. 병렬 루프 안에서 공유 데이터를 Write 하지 말것**

Concurrency Runtime 에서는 동기화를 위해 협동적 블러킹(Cooperative Blocking) 개념을 구현한 concurrency::critical\_section 클래스를 제공하며, 이는 Task가 가끔 공유 데이터에 접근할때 유용하게 사용될 수 있습니다. 하지만 병렬 루프 안에서 동기화가 빈번하게 이루어진다면 어떤 결과가 발생할까요? 아래에서는 병렬 루프 안에서 공유 데이터를 사용하여 소수의 합을 구하는 예제를 보여줍니다.

|  |
| --- |
| #include <ppl.h> #include <array> #include <numeric>  using namespace concurrency; using namespace std;  // 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;     **critical\_section cs;**      // 배열의 원소 중 모든 소수의 합을 계산합니다.     parallel\_for\_each(begin(a), end(a), [&](int i) {         **cs.lock();         prime\_sum += (is\_prime(i) ? i : 0);         cs.unlock();**     }); } |

이 예제에서는 공유 데이터인 prime\_sum 변수를 Write 하기 위해 빈번하게 동기화를 수행하게 되어 성능을 크게 저하시키고 결국엔 병렬 루프를 직렬화시키는 역할을 하게됩니다. 또한 concurrency::critical\_section 클래스는 협동적 블러킹을 사용하고 병렬 루프 안에서 많은 Task 들이 공유 데이터를 Write 하기 위해 블러킹되어 있기 때문에 단시간 안에 매우 많은 스레드가 생성되어 작업의 효율이 떨어지고 리소스 부족 현상이 나타날 수 있습니다. 이런 현상은 2번 항목에서 설명했던 내용과 일맥상통합니다.

일차적으로 병렬 루프 안에서 공유 데이터를 사용하는 것 자체가 좋은 패턴이 아니므로 이러한 상황에서는 공유 데이터 대신 Thread-Local Storage를 구현한 concurrency::combinable 클래스를 이용하여 동기화 없이 계산 작업을 수행하고 마지막에 계산 결과를 병합하는 방식으로 구현하는 것이 좋습니다. 그럼 위에서 구현된 병렬 루프를 방금 설명한 방식으로 수정해보도록 하죠.

|  |
| --- |
| **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>());** |

위와 같이 concurrency::combinable 클래스를 이용하면 스레드마다 고유한 복사본을 가지고 계산하기 때문에 스레드 동기화에 대한 오버헤드 없이 병렬 처리 작업을 수행할 수 있습니다.

**4.Task에서 사용하는 변수가 Task가 종료되기 전에 해제되지 않도록 주의할 것**

PPL에서 Task, Task 그룹 또는 병렬 알고리즘의 작업 함수를 구현하기 위해 람다 함수를 많이 사용합니다. 람다 함수에서는 외부 변수를 값 또는 참조 방식으로 캡쳐할 수 있는데 참조 방식으로 변수를 캡쳐할 경우에는 반드시 캡쳐한 변수의 수명이 Task가 종료될 때까지 남아있음을 보장해야합니다. 그럼 문제가 발생할 가능성이 있는 코드를 만들어보도록 하죠.

|  |
| --- |
| #include <ppl.h>  using namespace concurrency;  // 작업 수행을 위한 클래스 class object { public:     void action() const     {         // TODO: 작업 수행     } };  // 비동기 작업 실행을 위한 함수 void perform\_action(task\_group& tasks) {     // object 객체를 생성하여 작업을 비동기로 실행     **object obj;     tasks.run([&obj] {         obj.action();     });**      // NOTE: object 객체는 이 시점에서 해제되므로 task가 아직 실행중이라면     // 프로그램이 비정상 종료되거나 예상치 못한 문제가 발생할 수 있음 } |

perform\_action 함수 안에서 object 클래스 객체를 생성하고 이 객체를 이용하여 비동기로 작업을 수행합니다. **작업 함수에서는 object 객체를 참조 방식으로 캡쳐하여 사용하는데 만약 작업이 끝나기 전에 perform\_action 함수가 종료되어 object 객체가 해제되어버린다면 프로그램이 비정상 종료되거나 예상치 못한 문제가 발생할 수 있습니다.**

이러한 문제를 해결하기 위해여 object 객체를 참조 방식으로 캡쳐하는 대신 값으로 캡쳐하면 문제가 발생하지 않으며 이 경우에는 object 객체의 복사본을 가지게 됩니다.

|  |
| --- |
| // 비동기 작업 실행을 위한 함수 void perform\_action(task\_group& tasks) {     // object 객체를 생성하여 작업을 비동기로 실행     object obj;     **tasks.run([obj] {**         obj.action();     }); } |

하지만 action 메서드 내부에서 object 객체의 상태를 변경하고 그것이 원본에도 영향을 미쳐야 하는 상황이라면 위와 같은 방식으로는 해결할 수 없습니다. 그럼 이번엔 object 객체를 참조 방식으로 캡쳐하는 대신 비동기 작업이 종료될때까지 대기하는 방식으로 구현해보도록 하죠.

|  |
| --- |
| // 비동기 작업 실행을 위한 함수 void perform\_action(task\_group& tasks) {     // object 객체를 생성하여 작업을 비동기로 실행     object obj;     **tasks.run([&obj] {**         obj.action();     });      // 작업 종료 대기     **tasks.wait();** } |

이로서 object 객체가 비동기 작업이 종료될 때까지 존재한다는 것을 보장받을 수 있고 object 객체의 원본을 수정할 수도 있게 되었지만 작업 대기로 인하여 perform\_action 함수는 더이상 비동기로 동작하지 않게 되어버렸습니다.

이에 따른 방안으로 perform\_action 함수가 object 객체의 참조를 파라미터로 전달받고 함수를 호출하는 쪽에서 object 객체가 비동기 작업이 종료될때까지 존재하는 것을 보장하도록 하여 위에서 발생한 문제들을 해결할 수 있습니다.

|  |
| --- |
| // 비동기 작업 실행을 위한 함수 **void perform\_action(object& obj, task\_group& tasks)** {     // 전달받은 object 객체를 이용하여 작업을 비동기로 실행     tasks.run([&obj] {         obj.action();     }); } |