**5. 병렬 루프를 빠져나오려면 Cancellation 또는 Exception 을 이용할것**

PPL에서 Task 그룹 또는 병렬 알고리즘의 작업을 취소하기 위한 방법에는 크게 두가지가 있습니다. 한가지 방법은 Cancellation 메카니즘을 이용하는 것이고 다른 방법은 Exception 핸들링을 이용하는 것으로 이는 앞서 설명된 12장 강좌 에서 모두 설명되었던 내용입니다.

**병렬 작업 트리를 취소할 때에는 취소 상태를 부모로부터 자식들에게 전파되는 하향식 방법인 Cancellation 메카니즘을 이용하여 구현하면 효율적으로 작업을 취소할 수 있습니다.** 아울러 parallel\_for와 같은 병렬 알고리즘을 취소할 경우에도 부모 Task 그룹을 생성하고 병렬 알고리즘 내부에서 부모 Task 그룹을 취소하는 방식으로 깔끔하게 구현할 수 있습니다. 그럼 이에 대한 예제로 배열에 담긴 값을 병렬로 검색하는 parallel\_find\_any 함수를 작성해보도록 하죠.

|  |
| --- |
| #include <ppl.h> #include <iostream>  using namespace concurrency; using namespace std;  // 배열에서 검색하려는 값이 위치한 인덱스(찾지 못했을 경우에는 -1)를 반환 template<typename T> int parallel\_find\_any(const T a[], size\_t count, const T& what) {     // 검색 결과 인덱스를 -1 값으로 설정     int position = -1;      // 병렬 알고리즘이 부모 Task 그룹 tg 내부에서 동작하게함     **structured\_task\_group tg;**     **tg.run\_and\_wait([count, what, &a, &position, &tg]()**     {         **parallel\_for(std::size\_t(0), count, [what, &a, &position, &tg](int n) {**             if (a[n] == what)             {                 // 반환값을 설정하고 작업을 취소하여 병렬 루프를 빠져나옴                 position = n;                 **tg.cancel();**             }         });     });      return position; }  int wmain() {     // 정수 배열 생성     int numbers[] = { 3, 7, 11, 5, 2, 9, 8, 15, 13, 10 };     int index1, index2, num1 = 11, num2 = 1;      // 정수 11과 1에 해당하는 인덱스를 검색     index1 = parallel\_find\_any(numbers, sizeof(numbers) / sizeof(int), num1);     index2 = parallel\_find\_any(numbers, sizeof(numbers) / sizeof(int), num2);      // 검색 결과 출력     wcout << num1 << L"is at position " << index1 << endl;     wcout << num2 << L"is at position " << index2 << endl; }  // 실행결과  11 is at position 2 1 is at position -1 |

병렬 알고리즘은 내부적으로 Task 그룹을 사용하여 구현되어있고, 부모 Task 그룹의 작업을 취소하면 자식 Task 그룹의 작업도 취소된다는 특징 때문에 parallel\_for 루프 안에서 부모 Task 그룹을 취소하여 병렬 루프를 자연스럽게 빠져나올 수 있습니다. 이전에도 자세하게 설명드렸던 내용이기 때문에 이해하시는데 무리는 없으실 겁니다.

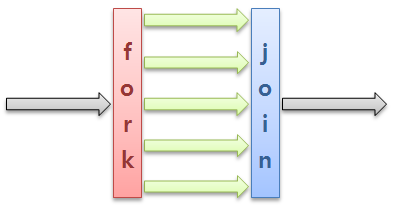
**Exception 핸들링을 이용한 병렬 작업의 취소는 Cancellation 메카니즘과는 반대로 상향식으로 동작하며 각각의 자식 Task 들은 독립적으로 작업을 취소하고 이를 부모에게 전파합니다.** 이러한 방식은 일반적으로 Cancellation 메카니즘에 비해서 그리 효율적이지는 않지만 때로는 Exception 핸들링을 사용하는 것이 더 적절한 경우도 있습니다. 지금부터는 Exception 핸들링 방식으로 병렬 알고리즘을 취소하는 예제를 작성해보도록 합시다. 먼저 트리 구조를 표현하는 tree 클래스가 있다고 가정하고, 모든 노드를 재귀적으로 탐색하여 작업 함수를 실행하게 하는 for\_all 메서드를 다음과 같이 작성해봅시다.

|  |
| --- |
| // 트리의 모든 노드를 재귀적으로 탐색하여 주어진 작업 함수를 실행 template<class Function> void tree::for\_all(Function& action) {     // 자식 노드에 대한 처리     **parallel\_for\_each(begin(\_children), end(\_children), [&](tree& child) {         child.for\_all(action);     });**      // 해당 노드에서 작업 함수를 호출     **action(\*this);** }      만약 for\_all 메서드를 호출하는 쪽에서 더이상 나머지 노드에 대한 작업 함수를 실행하지 않아도 된다고 판단한 경우에는 Exception을 발생시켜 간단하게 for\_all 메서드의 동작을 중단시킬 수 있습니다. 그럼 이번엔 for\_all 메서드를 사용하여 트리에서 주어진 값을 가지는 노드를 검색하는 search\_for\_value 함수를 작성해보도록 하겠습니다.    // 트리에서 주어진 값을 가지는 노드를 검색 template <typename T> void search\_for\_value(tree<T>& t, int value) {    **try** **{**         // 주어진 값을 검색하기 위해서 for\_all 함수를 호출         // 작업 함수에서는 값을 찾으면 이후 작업을 중단하기 위하여 Exception을 발생시킴         **t.for\_all([value](const tree<T>& node) {**            **if (node.get\_data() == value)**             **{                 throw &node;             }         });**    **}**     **catch (const tree<T>\* node)**    **{**         // 해당 값을 가지는 노드를 찾았을 경우         wstringstream ss;         ss << L"Found a node with value " << value << L'.' << endl;         wcout << ss.str();         return;     **}**      // 해당 값을 가지는 노드를 찾지 못하였을 경우     wstringstream ss;     ss << L"Did not find node with value " << value << L'.' << endl;     wcout << ss.str(); } |

search\_for\_value 함수에서는 for\_all 메서드를 호출하여 트리의 모든 노드에 대하여 작업 함수를 실행시키며 작업 함수에서는 현재 노드의 값이 검색할 값과 같은 경우에 해당 노드를 throw 시켜 예외를 발생함으로서 이후 작업을 중단시킴과 동시에 결과값을 전달할 수 있습니다.

**6. 병렬화를 가능한 레벨까지 최대한 끌어올릴것**

PPL을 이용하여, 특히 병렬 알고리즘을 사용하여 코드를 병렬화할 경우 우리는 알게 모르게 fork-join 구조를 사용하게됩니다. fork-join 구조란 아래 그림과 같이 하나의 작업을 여러개의 작은 세부 작업으로 나누고 모든 세부 작업이 종료될 때 까지 기다리는 것을 의미합니다. 그리고 각각의 세부 작업은 재귀적으로 자신을 또 다시 여러개의 세부 작업으로 나누어 처리할 수도 있죠.



<그림 1: fork-join 구조>

fork-join 구조가 여러가지 문제를 효율적으로 처리하는데 유용하게 사용될 수 있지만 한편으로는 동기화에 따른 오버헤드로 인해 성능을 감소시킬 수도 있습니다. 그럼 그 예로 이미지 프로세싱과 관련된 함수를 작성하면서 이를 설명하도록 하죠.

|  |
| --- |
| // 주어진 함수를 이용하여 비트맵 객체의 모든 픽셀을 처리 void ProcessImage(Bitmap\* bmp, const function<void (DWORD&)>& f) {     int width = bmp->GetWidth();     int height = bmp->GetHeight();      // 비트맵 객체 Lock     BitmapData bitmapData;     Rect rect(0, 0, bmp->GetWidth(), bmp->GetHeight());     bmp->LockBits(&rect, ImageLockModeWrite, PixelFormat32bppRGB, &bitmapData);      // 비트맵 포인터 얻기     DWORD\* image\_bits = (DWORD\*)bitmapData.Scan0;      // 비트맵의 각각의 픽셀에 대해 처리 함수 호출     for (int y = 0; y < height; ++y)     {         for (int x = 0; x < width; ++x)         {             // 현재 픽셀 값 얻기             DWORD\* curr\_pixel = image\_bits + (y \* width) + x;              // 처리 함수 호출             f(\*curr\_pixel);         }     }      // 비트맵 객체 Unlock     bmp->UnlockBits(&bitmapData); } |

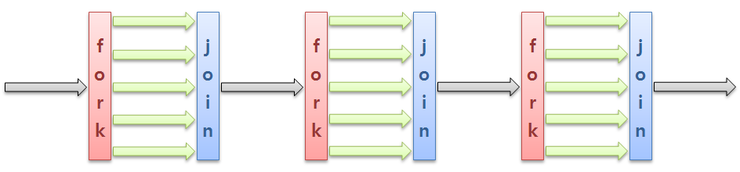
비트맵 객체와 처리 함수를 파라미터로 넘겨받아 모든 픽셀에 대해 처리 함수를 호출하는 함수입니다. 루프 안에서 처리하는 내용이 독립적이기 때문에 다음과 같이 parallel\_for 함수를 이용햐어 루프를 병렬화 할 수 있습니다.

|  |
| --- |
| // 주어진 함수를 이용하여 비트맵 객체의 모든 픽셀을 처리 void ProcessImage(Bitmap\* bmp, const function<void (DWORD&)>& f) {     int width = bmp->GetWidth();     int height = bmp->GetHeight();      // 비트맵 객체 Lock     BitmapData bitmapData;     Rect rect(0, 0, bmp->GetWidth(), bmp->GetHeight());     bmp->LockBits(&rect, ImageLockModeWrite, PixelFormat32bppRGB, &bitmapData);      // 비트맵 포인터 얻기     DWORD\* image\_bits = (DWORD\*)bitmapData.Scan0;      // 비트맵의 각각의 픽셀에 대해 처리 함수 호출     **parallel\_for (0, height, [&, width](int y)**     {         for (int x = 0; x < width; ++x)         {             // 현재 픽셀 값 얻기             DWORD\* curr\_pixel = image\_bits + (y \* width) + x;              // 처리 함수 호출             f(\*curr\_pixel);         }     });      // 비트맵 객체 Unlock     bmp->UnlockBits(&bitmapData); } |

외부 루프를 parallal\_for 함수를 이용하여 작업을 여러개의 세부 작업으로 나누어 처리하고 모든 세부 작업이 종료될때까지 기다리기 때문에 ProcessImage 함수는 fork-join 구조라고 할 수 있습니다. 그럼 이번엔 vector에 담긴 여러개의 비트맵 객체에 대하여 한 번씩 ProcessImage 함수를 호출하도록 하는 ProcessImages 함수를 작성해보도록 하죠.

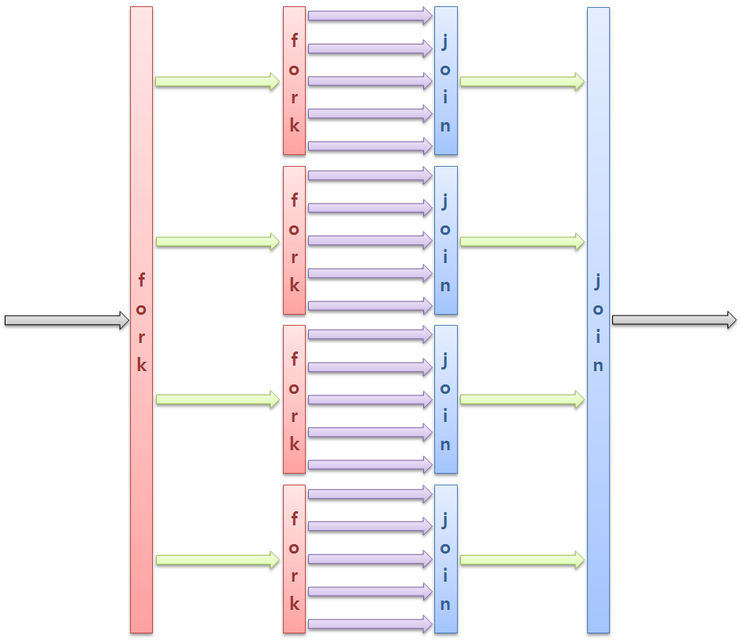
|  |
| --- |
| // vector에 포함된 모든 비트맵 객체에 대해 ProcessImage 함수를 호출 void ProcessImages(vector<Bitmap\*> bitmaps, const function<void (DWORD&)>& f) {     for\_each(begin(bitmaps), end(bitmaps), [&f](Bitmap\* bmp) {         ProcessImage(bmp, f);     }); } |

for\_each 루프 안에서 ProcessImage 함수를 호출할 때 마다 fork-join 작업이 발생하므로 이는 아래 그림과 같이 표현됩니다.

  
<그림 2: fork-join 작업의 반복>

위의 경우 처럼 fork-join 작업이 빈번하게 발생할 경우 이에 따른 스케줄링 오버헤드가 병렬 처리로 얻는 이득보다 커지는 상황이 발생할 수 있고 이 오버헤드는 프로세서의 수가 많아질 수록 더 증가합니다. 이러한 스케줄링 오버헤드는 다음과 같이 ProcessImages 함수에 구현된 for\_each 루프를 병렬화하여 병렬화 레벨을 최대한 끌어올림으로서 어느정도 줄일 수 있습니다.

|  |
| --- |
| // vector에 포함된 모든 비트맵 객체에 대해 ProcessImage 함수를 호출 void ProcessImages(vector<Bitmap\*> bitmaps, const function<void (DWORD&)>& f) {     **parallel\_for\_each(begin(bitmaps), end(bitmaps), [&f](Bitmap\* bmp) {**         ProcessImage(bmp, f);     }); } |

ProcessImage 함수를 반복하는 외부 루프까지 병렬화하였기 때문에 이는 아래 그림과 같이 표현되며 fork-join 작업이 재귀적으로 처리되는 것을 볼 수 있습니다.  
  


<그림 3: fork-join 작업의 재귀적 처리>

**7. Divide-and-Conquer 알고리즘에 parallel\_invoke 함수를 활용할것**

PPL을 활용하여 Divide-and-Conquer 알고리즘을 구현하려면 task\_group/structured\_task\_group 클래스, parallel\_invoke 함수 등을 사용할 수 있는데, 실행할 작업 개수가 고정되어있는 경우에는 parallel\_invoke 함수를 이용하면 좀 더 간결하게 코드를 작성할 수 있습니다. 그럼 이에 대한 예제로 parallel\_invoke 함수를 이용하여 정렬 알고리즘의 하나인 바이토닉 소트 알고리즘([Bitonic Sorting Algorithm, 위키백과 참조](http://en.wikipedia.org/wiki/Bitonic_sorting" \t "_blank))을 구현해보도록 하겠습니다.

|  |
| --- |
| template <class T> void parallel\_bitonic\_sort(T\* items, int lo, int n, bool dir) {     if (n > 1)     {         // 정렬할 데이터를 두 개의 파트로 분할하여 서로 다른 방향으로 정렬         int m = n / 2;         **parallel\_invoke(             [&] { parallel\_bitonic\_sort(items, lo, m, INCREASING); },             [&] { parallel\_bitonic\_sort(items, lo + m, m, DECREASING); }         );**          // 결과 병합         parallel\_bitonic\_merge(items, lo, n, dir);     } } |

parallel\_bitonic\_sort 함수에서는 parallel\_invoke 함수를 사용하여 재귀 호출을 병렬화하고 있음을 볼 수 있습니다. 이렇게 parallel\_invoke 함수를 사용하면 간단한 방법으로 Divide-and-Conquer 알고리즘에 병렬 프로그래밍을 적용할 수 있습니다.