**Parallel 정렬**

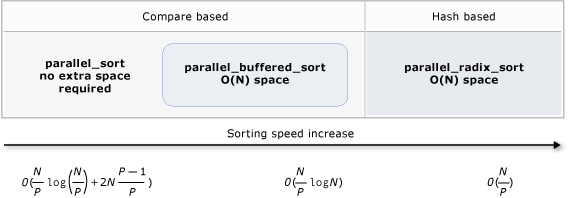
**PPL에서는 std::sort 함수를 대체할 수 있는 세 가지 버전의 병렬 처리 함수를 제공합니다.** 바로 **concurrency::parallel\_sort, concurrency::parallel\_buffered\_sort, concurrency::parallel\_radixsort** 함수인데요, 이 세 가지 정렬 함수는 매우 큰 데이터 셋을 정렬하는 경우 또는 정렬에 필요한 비교 구문이 복잡하여 시간이 오래 소요되는 경우에 아주 유용하며 성능 향상에 큰 도움이 됩니다.

먼저 parallel\_sort 함수와 parallel\_buffered\_sort 함수는 std::sort 함수와 마찬가지로 비교 기반의 정렬 알고리즘을 사용합니다. 컨테이너 원소들의 값을 비교하여 정렬을 수행하게 되죠. **parallel\_sort 함수는 정렬을 수행하는데 필요한 추가적인 메모리가 필요하지 않은 반면에 parallel\_buffered\_sort 함수는 O(N)의 추가적인 메모리를 필요로 하지만 보통 parallel\_sort 함수보다 좋은 성능을 보입니다.** 두 함수는 std::sort 함수와 사용 방식이 일치하기 때문에 기존에 작성되었던 코드도 아주 쉽게 병렬 방식으로 대체할 수 있습니다.

**위의 두 함수와는 달리 parallel\_radixsort 함수는 해시 기반의 정렬 알고리즘을 사용하고 해시 함수는 정수 타입을 반환해야 합니다.** 또한 parallel\_radixsort 함수는 parallel\_buffered\_sort 함수와 마찬가지로 O(N)의 추가적인 메모리를 필요로 합니다. 다음 표에서는 방금 설명드렸던 세 가지 함수의 특징을 비교하여 보여줍니다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **알고리즘** | **설명** | **정렬 방식** | **메모리 사용** | **시간 복잡도** |
| **parallel\_sort** | 일반적인 목적의 정렬 | 비교 기반 | 없음 | O((N/P)log(N/P) + 2N((P-1)/P)) |
| **parallel\_buffered\_sort** | 일반적인 목적의 빠른 정렬 | 비교 기반 | O(N) | O((N/P)log(N)) |
| **parallel\_radixsort** | 정수 키 기반의 빠른 해시 정렬 | 해시 기반 | O(N) | O(N/P) |

그리고 아래에는 각각의 함수의 특징과 성능을 그림으로 잘 표현하고 있습니다. parallel\_radixsort 함수는 O(N)의 추가적인 메모리 공간을 필요로 하고 해시 함수의 성능에 따라 차이가 있기는 하지만 일반적으로 가장 좋은 성능을 보여주네요.



<그림 원본: <http://msdn.microsoft.com/en-us/library/dd470426.aspx>>

**Parallel 정렬 함수는 Move Semantics를 지원합니다.**만약 컨테이너의 원소 타입이 Move 대입 연산자를 제공한다면 정렬 수행 도중 각각의 원소를 swap할 때 Copy 대입 연산자 대신 Move 대입 연산자를 사용하게 되어 성능 향상에 도움이 됩니다. 혹시 Move 대입 연산자가 무엇인지 아직 모르시는 분들은 이전에 포스팅한 '[[C++11] Rvalue Reference 강좌](http://devmachine.blog.me/178065606)'를 참고하시기 바랍니다. 그리고 이후에 나오는 모든 예제에서는 난수 생성기로 mt19937을 사용하는데 이것에 내용은 역시 이전에 포스팅한 ['[C++] 랜덤 함수, 이제는 바꿔보자!! mt19937'](http://devmachine.blog.me/177685246) 강좌를 참고하세요.

그럼 지금부터 본격적으로 예제와 함께 진행하도록 하겠습니다. ^^ 먼저 간단하게 정수 값을 정렬하는 parallel\_sort 함수부터 살펴보도록 하죠.

|  |
| --- |
| #include <ppl.h> #include <random> #include <iostream>  using namespace concurrency; using namespace std;  int wmain() {     // 매우 큰 크기의 벡터에 랜덤한 정수값 생성     vector<int> values(25000000);     **generate(begin(values), end(values), mt19937(42));**      // 오름 차순으로 정렬     **parallel\_sort(begin(values), end(values));**      // 최소값, 중간값, 최대값 출력     wcout << "V(0)        = " << values[0] << endl;     wcout << "V(12500000) = " << values[12500000] << endl;     wcout << "V(24999999) = " << values[24999999] << endl; }  // 실행결과  V(0)               = -2147483129 V(12500000) = -427327 V(24999999) = 2147483311 |

매우 큰 크기의 정수 벡터에 랜덤한 값을 생성해놓고 오름 차순으로 정렬하는 예제입니다. parallel\_sort 함수의 세 번째 파라미터는 비교 함수이며 이번 예제와 같이 생략되었을 경우에는 std::less 를 사용합니다. std::sort 함수를 한 번이라도 사용해보셨던 분들이라면 이해하는데 전혀 어려움이 없으실겁니다. 그럼 다음으로 parallel\_buffered\_sort 함수를 이용하여 복소수를 정렬하는 예제를 살펴보도록 하죠.

|  |
| --- |
| #include <ppl.h> #include <random> #include <iostream> #include <complex>  using namespace concurrency; using namespace std;  int wmain() {     // 매우 큰 크기의 벡터에 랜덤한 복소수 생성     vector<complex<double>> values(25000000);     generate(begin(values), end(values), mt19937(42));      // 오름 차순으로 정렬     **parallel\_buffered\_sort(begin(values), end(values),         [](const complex<double>& left, const complex<double>& right) {             return left.real() < right.real();     });**      // 최소값, 중간값, 최대값 출력     wcout << "V(0)        = " << values[0] << endl;     wcout << "V(12500000) = " << values[12500000] << endl;     wcout << "V(24999999) = " << values[24999999] << endl; }  // 실행결과  V(0)               = (383,0) V(12500000) = (2.1479e+009,0) V(24999999) = (4.29497e+009,0) |

이번엔 정수 대신 std::complex 클래스를 이용하여 랜덤한 복소수 값을 생성하였고, parallel\_sort 함수 대신 parallel\_buffered\_sort 함수를 이용하였습니다. 그리고 std::complex 클래스는 비교 연산자를 제공하지 않기 때문에 람다 함수로 복소수의 실수부를 비교하는 함수를 구현하여 세 번째 파라미터로 전달하여 줍니다. 전혀 어렵지 않죠? ^^ 그럼 계속해서 parallel\_radixsort 예제까지 쭉쭉 진행해봅시다.

|  |
| --- |
| #include <ppl.h> #include <random> #include <iostream>  using namespace concurrency; using namespace std;  // 3차원 포인트 struct Point {     int X;     int Y;     int Z; };  // 두 포인트의 유클리안 거리를 구하는 함수 size\_t euclidean\_distance(const Point& p1, const Point& p2) {     int dx = p1.X - p2.X;     int dy = p1.Y - p2.Y;     int dz = p1.Z - p2.Z;     return static\_cast<size\_t>(sqrt((dx\*dx) + (dy\*dy) + (dz\*dz))); }  int wmain() {     // 거리를 계산할 중심점     const Point center = { 3, 2, 7 };      // 랜덤한 포인트 생성     vector<Point> values(7);     mt19937 random(42);     generate(begin(values), end(values), [&random] {         Point p = { random()%10, random()%10, random()%10 };         return p;     });      // 생성된 포인트 출력     wcout << "Before sorting:" << endl;     for\_each(begin(values), end(values), [center](const Point& p) {         wcout << L'(' << p.X << L"," << p.Y << L"," << p.Z             << L") D = " << euclidean\_distance(p, center) << endl;     });     wcout << endl;      // 중심점으로부터의 거리를 기준으로 정렬     **parallel\_radixsort(begin(values), end(values),         [center](const Point& p) -> size\_t {             return euclidean\_distance(p, center);     });**      // 정렬된 포인트 출력     wcout << "After sorting:" << endl;     for\_each(begin(values), end(values), [center](const Point& p) {         wcout << L'(' << p.X << L"," << p.Y << L"," << p.Z             << L") D = " << euclidean\_distance(p, center) << endl;     });     wcout << endl; }    // 실행결과  Before sorting: (2,7,6) D = 5 (4,6,5) D = 4 (0,4,0) D = 7 (3,8,4) D = 6 (0,4,1) D = 7 (2,5,5) D = 3 (7,6,9) D = 6  After sorting: (2,5,5) D = 3 (4,6,5) D = 4 (2,7,6) D = 5 (3,8,4) D = 6 (7,6,9) D = 6 (0,4,0) D = 7 (0,4,1) D = 7 |

parallel\_radixsort 함수는 세 번째 파라미터로 해시 함수를 요구하기 때문에(생략할 경우 std::hash 사용) 위 예제에서는 euclidean\_distance 함수를 호출하는 람다 함수를 구현하여 해시 함수로 사용하였습니다. 물론 이것은 다음과 같은 해시 함수 오브젝트 형태로 구현할 수도 있습니다.

|  |
| --- |
| // 두 포인트의 거리를 구하는 Functor 클래스 class hash\_distance { public:     hash\_distance(const Point& reference)         : m\_reference(reference)     {     }      size\_t operator()(const Point& pt) const {         return euclidean\_distance(pt, m\_reference);     }  private:     Point m\_reference; };    // Functor 클래스 hash\_distance를 이용하여 정렬 parallel\_radixsort(begin(values), end(values), hash\_distance(center)); |

**정렬 알고리즘 선택**

지금까지 PPL에서 제공하는 세 가지 정렬 함수의 특징과 사용법에 대하여 알아보았습니다. 이제 사용 방법에 대해서는 알게 되었지만 내가 구현하는 코드에서 어떠한 알고리즘을 선택하여 구현해야 가장 효율적인지에 대한 고민이 생깁니다. 과연 우리는 어떠한 기준으로 병렬 알고리즘을 선택하야 할까요? 일반적으로는 다음과 같이 나열된 가이드라인에 의해 알고리즘을 선택할 수 있습니다.

* 1,000개 이하의 작은 데이터셋을 정렬할 경우에는 병렬 알고리즘 대신 std::sort 함수를 사용한다.
* 10,000 ~ 100,000개의 데이터셋을 정렬할 때 메모리를 아껴야하는 상황 또는 메모리 할당시 동기화가 필요한 경우 parallel\_sort 함수를 사용한다.
* 10,000 ~ 100,000개의 데이터셋을 정렬할 때 O(N) 크기의 메모리 공간의 여유가 있고 컴퓨팅 자원이 충분하거나 비교 연산이 복잡한 경우 parallel\_buffered\_sort 함수를 사용한다.
* 100,000개 이상의 큰 데이터셋을 정렬할 때 O(N) 크기의 메모리 공간의 여유가 있고 비교 연산이 해시 연산보다 복잡한 경우 또는 두 연산 모두 복잡한 경우 parallel\_radixsort 함수를 사용한다.

하지만 위에서 언급한 사항은 어디까지나 가이드라인일 뿐입니다. **정렬 알고리즘을 선택하는 가장 좋은 방법은 주어진 시나리오에서 여러 가지 알고리즘으로 정렬 작업을 시뮬레이션 해보고 가장 좋은 성능을 내는 알고리즘을 선택하는 것입니다.** 그리고 데이터셋의 개수에 따라 다른 정렬 알고리즘을 사용하는것도 하나의 방법이죠. 이제 마지막으로 천만개의 정수 데이터셋을 생성하고 4가지 정렬 알고리즘을 사용하여 각각 소요되는 시간을 출력하는 예제를 작성해보도록 합시다.

|  |
| --- |
| #include <ppl.h> #include <random> #include <iostream> #include <windows.h>  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; }  const size\_t DATASET\_SIZE = 10000000;   // 일정한 패턴의 랜덤한 정수값을 가지는 데이터셋을 생성하여 반환합니다. vector<size\_t> GetData() {     vector<size\_t> data(DATASET\_SIZE);     generate(begin(data), end(data), mt19937(42));     return data; }  int wmain() {     // std::sort 함수를 이용한 정렬     auto data = GetData();     wcout << L"Testing std::sort...";     auto elapsed = time\_call([&data] { sort(begin(data), end(data)); });     wcout << L" took " << elapsed << L" ms." <<endl;      // concurrency::parallel\_sort 함수를 이용한 정렬     data = GetData();     wcout << L"Testing concurrency::parallel\_sort...";     elapsed = time\_call([&data] { parallel\_sort(begin(data), end(data)); });     wcout << L" took " << elapsed << L" ms." <<endl;      // concurrency::parallel\_buffered\_sort 함수를 이용한 정렬     data = GetData();     wcout << L"Testing concurrency::parallel\_buffered\_sort...";     elapsed = time\_call([&data] { parallel\_buffered\_sort(begin(data), end(data)); });     wcout << L" took " << elapsed << L" ms." <<endl;      // concurrency::parallel\_radixsort 함수를 이용한 정렬     data = GetData();     wcout << L"Testing concurrency::parallel\_radixsort...";     elapsed = time\_call([&data] { parallel\_radixsort(begin(data), end(data)); });     wcout << L" took " << elapsed << L" ms." <<endl; }  // 실행결과  Testing std::sort... took 1016 ms. Testing concurrency::parallel\_sort... took 203 ms. Testing concurrency::parallel\_buffered\_sort... took 172 ms. Testing concurrency::parallel\_radixsort... took 47 ms. |

위에서 출력된 결과 값은 실제로  제 PC 환경(Intel Core i7 CPU, 8GB RAM)에서 직접 실행한 결과입니다. 역시 예상했던 대로 parallel\_radixsort 함수의 정렬 속도가 가장 빨랐고 std::sort 함수와 비교했을 때 무려 22배 정도 빠른 속도를 보여주었습니다. 단순히 함수 하나 바꿨을 뿐인데...엄청난 성능 향상이 발생하네요.