C++11 ラムダ式により並列化されたループの定型コードを軽減する

この記事は、インテル® デベロッパー・ゾーンに公開されている「Reducing boilerplate code in parallelized loops with C++11 lambda expressions」の日本語参考訳です。

C++11 標準 (旧称 C++0x) でラムダ式が導入され、定型コードが大幅に軽減されました。インテル® C++ コンパイラーでは、インテル® スレッディング・ビルディング・ブロック (インテル® TBB) を使用してループを並列化する場合、ラムダ式を利用して定型コードを減らすことができます。

昨年、非常に似たシナリオの多くのプロジェクトに取り組む機会がありました。それらはパフォーマンスを向上する必要があり、多くの驚異的な並列ワークロードが主な hotspots となっていました。パフォーマンスを大幅に向上するには、ベクトル化の利点が得られるようにループの並列化が必要でした。しかし、すべてのプロジェクトで同じ状況に陥りました: インテル® TBB を使用してループを並列化するために必要な定型コードに対して、開発者が苦言を呈したのです。長年 C++11 コードで開発を行い、インテル® TBB を使用するコードの記述には C++11 ラムダを利用してきた私にとって、これは驚きでした。

多くの開発者は数年前に C++ から遠ざかり、最近また C++ に戻ってきましたが、今では不要となった古い方法でコードを記述し、コードの可読性、理解、保守を困難にしていました。インテル® C++ コンパイラーは、バージョン 11 で C++11 機能のサポートを追加して以来、ラムダ式を含む多くの機能にサポートを拡張してきました。

ラムダ式は、ファンクター構文を簡潔に指定する方法です。そのため、ファンクターの代わりにラムダ式を使用できます。次の行は、C++11 ラムダ式の基本構文と要素です。

```
[ captures ] (parameters) -> returnTypesDeclaration { lambdaStatements; }
```

[captures]: ラムダ式を開始する、capture 節 (ラムダ導入子とも呼ばれる) です。ラムダ式で利用可能な外部変数と 値渡し (コピー) か、参照渡しかを指定します。キャプチャー・デフォルトは多数あり、capture 節はラムダ式の開始を簡単に識別できるようにします。

(parameters): 引数リストを指定します (オプション)。ラムダ宣言子とも呼ばれます。関数がゼロ引数 (引数なしの関数) の場合、引数リストは省略できます。

-> returnTypeDeclaration: ラムダ式の戻り型を指定します (オプション)。

{ lambdaStatements; }: ラムダ式の本体です。ラムダ本体内に記述する構文はすべて、キャプチャー変数と引数を利用できます。

空の capture 節は **[ ]** で宣言され、ラムダが何もキャプチャーしないこと (つまり、ラムダ式の本体が囲まれたスコープで変数にアクセスしないこと) を示します。ほとんどの場合、並列化されたループでは、処理する番号を取得するため、少なくとも 1 つの変数にアクセスする必要があります。**[ ]** キャプチャーは、並列化されたループに含まれるラムダ式でよく見られます。

[=] キャプチャーは、ラムダ内で参照される変数を値渡し (コピー) します。ラムダ内で参照される外部変数は、自動的に値渡しされます。[=] キャプチャーは、並列化されたループに含まれるラムダ式でよく見られます。ただし、多くのケースでは、ラムダで変数のキャプチャーが不要な場合、**[ ]** に置き換えることができます。

[&] キャプチャーは、ラムダ内で参照される変数を参照渡しします。ラムダ内で参照される外部変数は、自動的に参照渡しされます。

デフォルト・キャプチャーを例外付きで使用することもできます。例えば、**[=, &number]** は、number 変数を除くラムダ内で参照される値をすべて値渡し (コピー) します。number 変数は参照渡しされます。

number 引数を受け取り、number に関連する CPU 負荷の高いコードを実行する、次の関数宣言コードについて考えてみます。この関数は空の本体を宣言していますが、並列化されたループの各 number (0 ～ 1023) で呼び出されます。

```
#include <iostream>

#include "tbb/tbb.h"

using namespace tbb;

const size_t numbers_to_process = 1024;

void process_number(size_t number) {
    // number で受け取った値を処理する関数
}
```

次の行は、C++11 のラムダ式を使用して、**0 ～ numbers_to_process - 1** (1023) の範囲で **process_number** メソッドを呼び出す並列化されたループを簡単に表現しています。

```
int main()

{
    parallel_for(
        blocked_range<size_t>(0, numbers_to_process),
        [](const blocked_range<size_t>& r) {

        for (size_t i = r.begin(); i != r.end(); ++i) {
            process_number(i);
                        }
        });

        std::cin.ignore();

        return 0;
}
```

次のコードは、**parallel_for** (**tbb::parallel_for)** で指定されたラムダ式を宣言します。

```
[](const blocked_range<size_t>& r) {
    for (size_t i = r.begin(); i != r.end(); ++i) {
        process_number(i);
}
```

このコードは非常に分かりやすいです。ラムダ式は変数をキャプチャーしないため、**[ ]** で始まります。ラムダは、**blocked_range<size_t> &r** を引数として受け取り、受け取った範囲を **i** の開始値・終了値とし、**i** を引数として **process_number** メソッドを呼び出すシーケンシャル for ループを実行します。

次の行を使用することもできます。**parallel_for** に含まれる多くのラムダ式では、変数をキャプチャーしないにもかかわらず、**[=]** が使用されることがあります。しかし、変数をキャプチャーしない場合は、空の capture 節 [ ] を使用すべきです。

```
[=](const blocked_range<size_t>& r) {

    for (size_t i = r.begin(); i != r.end(); ++i) {

        process_number(i);

}
```

自動チャンクを使用し、整数の単一次元の反復のみで、連続する整数範囲をループする場合、さらに簡潔なコードを記述できます。

次の行は、C++11 のラムダ式とインテル® TBB のコンパクト形式を使用して、**0 ～ numbers_to_process - 1** (1023) の範囲で **process_number** メソッドを呼び出す並列化されたループを簡単に記述します。

```
int main()
{
    parallel_for(size_t(0), numbers_to_process, [](size_t i) {
        process_number(i);
    });

    std::cin.ignore();

    return 0;
}
```

次のコードは、**parallel_for** (**tbb::parallel_for)** で指定されたラムダ式を宣言します。

```
[](size_t i) { process_number(i); });
```

このコードは非常に分かりやすいです。ラムダ式は変数をキャプチャーしないため、**[ ]** で始まります。ラムダは、**size_t i** を引数として受け取り、**i** を引数として **process_number** メソッドを呼び出すだけです。

次の行を使用することもできます。**parallel_for** に含まれる多くのラムダ式では、変数をキャプチャーしないにもかかわらず、**[=]** が使用されることがあります。しかし、変数をキャプチャーしない場合は、空の capture 節 [ ] を使用すべきです。

```
[=](size_t i) { process_number(i); });
```

C++11 のラムダ式を使用して、インテル® TBB で並列化されたループは、シーケンシャル for ループと同じぐらい簡単に理解できます。最近のインテル® プロセッサーを利用する現代化されたコードを記述する場合は、インテル® C++ コンパイラーのすべての機能を利用して定型コードを軽減することも忘れないでください。

いったん、C++11 ラムダを使い始めたら、戻ることはできなくなるでしょう。

参考リンク:
インテル® C++ コンパイラーでサポートされる C++11 の機能