Portfolio 7 - Intel TBB

Sam Bowyer

2023-05-29

Parallelisation in C++ with Intel TBB

Introduction

In the previous portfolio we looked at the basics of parallelisation in C++ using the OpenMP library. In this portfolio we will look at the Intel Threading Building Blocks (TBB) library. TBB is another library for parallelisation in C++ and is very similar to OpenMP. It is also very widely used and is a good alternative to OpenMP.

In particular, this portfolio will focus on the use of `map` and `reduce` operations in TBB. These are two very common operations in parallel programming and are very useful for parallelising algorithms. However, before we cover these we will also introduce the concept of lambda functions in C++.

Lambda functions

Lambda functions were introduced in C++11 and allow us to define a function without giving it a name. They are very useful for defining simple functions that are only used once, for example, we can define a lambda function that adds two numbers together as follows:

```
[](int x, int y) { return x + y; }
```

This might be used in a larger program named `lambdaExample.cpp` as follows:

```
#include <iostream>

int main(int argc, char **argv)
{
    for (int i=1; i<=10; ++i)
    {
        std::cout << "i == " << i << std::endl;

        auto func = [&](int x){ return x + i; };

        std::cout << "func(5) equals " << func(5) << std::endl;
    }

    return 0;
}
```

Below we then compile and run this program:

```
g++ -std=c++11 lambdaExample.cpp -o lambdaExample
./lambdaExample
```

```
## i == 1
## func(5) equals 6
```

```
## i == 2
## func(5) equals 7
## i == 3
## func(5) equals 8
## i == 4
## func(5) equals 9
## i == 5
## func(5) equals 10
## i == 6
## func(5) equals 11
## i == 7
## func(5) equals 12
## i == 8
## func(5) equals 13
## i == 9
## func(5) equals 14
## i == 10
## func(5) equals 15
```

We can define the lambda function with different levels of closure based on what we put inside the `[]` brackets. In particular, we can put the following inside the `[]` brackets:

- `&` - Capture all variables by reference
- `=` - Capture all variables by value
- `x` - Capture the variable `x` by value
- `&x` - Capture the variable `x` by reference

By default, if we do not specify anything inside the `[]` brackets then the lambda function will capture all variables by value.

Map

It is very common in parallel programming to want to apply a function to each element of a container. For example, we might want to apply a function to each element of a vector. This is known as a `map` operation.

We can write an example map implementation in C++ as follows:

```cpp
#include <iostream>
#include <vector>

template <typename T, typename F>
std::vector<T> map(const std::vector<T> &input, F func)
{
    std::vector<T> output(input.size());

    for (size_t i=0; i<input.size(); ++i)
    {
        output[i] = func(input[i]);
    }

    return output;
}

int main(int argc, char **argv)
{
    std::vector<int> input = {1, 2, 3, 4, 5};
```

```
    auto output = map(input, [](int x){ return x * x; });

    for (auto x : output)
    {
        std::cout << x << std::endl;
    }

    return 0;
}
```

In this case, we are mapping over the vector $[1, 2, 3, 4, 5]$ and squaring each element. We can then compile and run this program as follows:

```
g++ -std=c++11 mapExample.cpp -o mapExample
./mapExample
```

```
## 1
## 4
## 9
## 16
## 25
```

Reduce

Reduce is another very common operation in parallel programming. It is used to combine the elements of a container into a single value. For example, we might want to sum all the elements of a vector. This is known as a `reduce` operation.

We can write an example reduce implementation in C++ as follows:

```
#include <iostream>
#include <vector>

template <typename T, typename F>
T reduce(const std::vector<T> &input, F func, T init)
{
    T output = init;

    for (size_t i=0; i<input.size(); ++i)
    {
        output = func(output, input[i]);
    }

    return output;
}

int main(int argc, char **argv)
{
    std::vector<int> input = {1, 2, 3, 4, 5};

    auto output = reduce(input, [](int x, int y){ return x + y; }, 0);

    std::cout << output << std::endl;

    return 0;
}
```

Note that we not only have to supply our function `func` to reduce two items into one, but also an initial value `init`. It is important to consider your initial value carefully, for instance if `func` is addition we would probably want `init=0`, but if `func` is instead multiplication we would more likely want `init=1` (since 0 is the identity element for addition and 1 is the identity element for multiplication).

Compiling and running this program may be done as follows:

```
g++ -std=c++11 reduceExample.cpp -o reduceExample
./reduceExample
```

```
## 15
```

Parallel map and reduce

A parallel implementation of `map` and `reduce` can be implemented using the TBB library. TBB provides a `parallel_for` function that can be used to parallelise a `for` loop, taking a `blocked_range` of values to iterate over (each block inside this range may contain more than one item, but generally it will be a small number of items per block) and a lambda function to apply to (each item inside) each block. The lambda function is executed in parallel on multiple threads: one thread per block. The `parallel_for` function will wait until all the threads have finished executing the lambda function before returning.

The `parallel_for` function can be used to implement a parallel `map` operation as follows (again we are using `map` to square each element in $[1, 2, 3, 4, 5]$):

```
#include <iostream>
#include <vector>
#include <tbb/parallel_for.h>

template <typename T, typename F>
std::vector<T> map(const std::vector<T> &input, F func)
{
    std::vector<T> output(input.size());

    tbb::parallel_for(tbb::blocked_range<int>(0,input.size()), [&](tbb::blocked_range<int> r)
    {
        for (int i=r.begin(); i<r.end(); ++i)
        {
            output[i] = func(input[i]);
        }
    });

    return output;
}

int main(int argc, char **argv)
{
    std::vector<int> input = {1, 2, 3, 4, 5};

    auto output = map(input, [](int x){ return x * x; });

    for (auto x : output)
    {
        std::cout << x << std::endl;
    }

    return 0;
```

```
}
```

This can then be compiled and run as follows (note that we are including the flag `ltbb` to tell our compiler to use the TBB library, and also the `-O3` flag to optimize our code):

```
g++ --std=c++14 -O3 parallelMapExample.cpp -ltbb -o parallelMapExample
./parallelMapExample
```

```
## 1
## 4
## 9
## 16
## 25
```

A parallel implementation of `reduce` is given in the TBB library: `parallel_reduce`. This has the form

```
tbb::parallel_reduce(
    tbb::blocked_range r,
    initial_value,
    [](tbb::blocked_range<int> r, double running_total){OPERATIONS IN HERE},
    reduce_function
)
```

Here we again must provide a `blocked_range`, initial value and `reduce_function`, but as the third argument we also have to provide a function that defines what the program shoulod do for each chunk. To obtain a 'pure' reduce function that sums over a vector, we will set this lambda function to simply add the elements in each block and use the `std::plus<double>` function as our `reduce_function`. (We also use our `parallel_for` code to generate the vector of values that we're summing over.)

```
#include <iostream>
#include <vector>
#include <cmath>
#include <algorithm>

#include <tbb/parallel_for.h>
#include <tbb/parallel_reduce.h>

int main(int argc, char **argv)
{
    auto values = std::vector<double>(10000);

    // Generate values to sum over
    tbb::parallel_for( tbb::blocked_range<int>(0,values.size()),
                       [&](tbb::blocked_range<int> r)
    {
        for (int i=r.begin(); i<r.end(); ++i)
        {
            values[i] = std::sin(i * 0.001);
        }
    });

    // Sum the values using parallel_reduce
    auto total = tbb::parallel_reduce(
                    tbb::blocked_range<int>(0,values.size()),
                    0.0,
                    [&](tbb::blocked_range<int> r, double running_total)
                    {
```

```
                        for (int i=r.begin(); i<r.end(); ++i)
                        {
                            running_total += values[i];
                        }
                        return running_total;
                    }, std::plus<double>() );

    std::cout << total << std::endl;

    return 0;
}
```

We compile and run this as follows:

```
g++ --std=c++14 -O3 parallelReduceExample.cpp -ltbb -o parallelReduceExample
./parallelReduceExample
```

```
## 1839.34
```

Parallel Mapreduce

It is often very useful to combine the `map` and `reduce` operations into a single operation: mapping a function over a collection and then reducing the output of that map operation using another function. This is known as a `mapreduce` operation and can be used to achieve the same result we got in the previous example by combining the value generation loop (with `parallel_for`) and the value summing loop (with `parallel_reduce`).

Below we show a generic parallel `mapreduce` implementation using Intel-TBB's `parallel_for` and `parallel_reduce` functions, which takes in one function for mapping (here we use a lambda function to square each item), a function for reducing (we use `std::plus<int>`) and an array to `mapreduce` over (here, the vector $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$).

```
#include <iostream>
#include <vector>
#include <algorithm>

#include <tbb/parallel_for.h>
#include <tbb/parallel_reduce.h>

template<class MAPFUNC, class REDFUNC>
auto mapReduce(MAPFUNC mapfunc, REDFUNC redfunc,
               const std::vector<int> &arg1)
{
    return tbb::parallel_reduce(
              tbb::blocked_range<int>(0,arg1.size()),
              0,
              [&](tbb::blocked_range<int> r, int running_total)
              {
                  for (int i=r.begin(); i<r.end(); ++i)
                  {
                      running_total = redfunc(running_total,
                                              mapfunc(arg1[i]) );
                  }

                  return running_total;
              }, redfunc );
}
```

```
int main(int argc, char **argv)
{
    auto a = std::vector<int>( { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 } );

    auto result = mapReduce( [](int x){ return x * x; },
                             std::plus<int>(), a );

    std::cout << result << std::endl;

    return 0;
}
```

```
g++ --std=c++14 -O3 parallelMapReduceExample.cpp -ltbb -o parallelMapReduceExample
./parallelMapReduceExample
```

```
## 385
```