Intel TBB Portfolio

Rachel Wood

2023-04-25

Intel thread building blocks (TBB) allows you to make use of multicore processors in C++, building off the core ideas of functional and parallel functional programming.

Here we make use of a workshop folder which is downloaded at this link.

Functional Programming

Functions as Variables

This framework of programming lets us treat functions as any other variables, and so they can be passed as parameters and returned as outputs of other functions.

For example, in the following file, we set the variable `a` to be the `sum()` function:

```
#include <iostream>

int sum(int x, int y)
{
    return x + y;
}

int main(int argc, char **argv)
{
    auto a = sum;
    auto result = a(3, 7);
    std::cout << result << std::endl;
    return 0;
}
```

we can now compile and run this code to get the output `10`:

```
g++ --std=c++14 sum.cpp -o sum
./sum
10
```

so the code of a function is passed to a new variable as you would a floating point number or a string.

Similarly we can pass them as arguments in a function:

```cpp
#include <iostream>

int sum(int x, int y)
{
    return x + y;
}

int difference(int x, int y)
{
    return x - y;
}

template<class FUNC, class ARG1, class ARG2>
auto call_function(FUNC func, ARG1 arg1, ARG2 arg2)
{
    std::cout << "Calling a function with arguments " << arg1
              << " and " << arg2;

    auto result = func(arg1,arg2);

    std::cout << ". The result is " << result << std::endl;

    return result;
}

int main(int argc, char **argv)
{
    auto result = call_function( difference, 9, 2 );
    std::cout << result << std::endl;

    result = call_function( sum, 3, 7 );
    std::cout << result << std::endl;

    return 0;
}
```

when compiled and run we get the output

```
Calling a function with arguments 4 and 5. The result is 20
20
```

Mapping

This is a a concept that allows us to create a wrapper for a function that allows the same function to be applied to multiple sets of data. For example the following allows us to create a `map()` function, which creates a map from a function taking two arguments:

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

template<class FUNC, class T>
auto map(FUNC func, const std::vector<T> &arg1, const std::vector<T> &arg2)
```

```
{
    int nvalues = std::min( arg1.size(), arg2.size() );

    auto result = std::vector<T>(nvalues);

    for (int i=0; i<nvalues; ++i)
    {
        result[i] = func(arg1[i], arg2[i]);
    }

    return result;
}

int multiply(int x, int y)
{
    return x * y;
}

template<class T>
void print_vector(const std::vector<T> &values)
{
    std::cout << "[";

    for (const T &value : values)
    {
        std::cout << " " << value;
    }

    std::cout << " ]" << std::endl;
}

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

    result = map( multiply, a, b );
    print_vector(result);

    return 0;
}
```

We can generalise this map function to accept more than two arguments:

```
template<class FUNC, class... ARGS>
auto map(FUNC func, const std::vector<ARGS>&... args)
{
    typedef typename std::result_of<FUNC(ARGS...)>::type RETURN_TYPE;

    int nargs=detail::get_min_container_size(args...);

    std::vector<RETURN_TYPE> result(nargs);
```

```
    for (size_t i=0; i<nargs; ++i)
    {
        result[i] = func(args[i]...);
    }

    return result;
}
```

Example: The `shakespeare` folder contains the full text of many Shakespeare plays. We create a C++ program which counts the number of lines in each of these plays, which includes the a header files: - `include/part1.h`: contains the `map()` function given above - `include/filecounter.h`: contains the `count_lines()` function to be mapped and the `get_arguments()` function to obtain the fileanames: We store the following code in a file called `countlines.cpp`

```
#include "workshop/include/part1.h"
#include "workshop/include/filecounter.h"

using namespace part1;
using namespace filecounter;

int main(int argc, char **argv)
{
    auto filenames = get_arguments(argc, argv);

    auto results = map( count_lines, filenames );

    for (size_t i=0; i<filenames.size(); ++i)
    {
        std::cout << filenames[i] << " = " << results[i] << std::endl;
    }

    return 0;
}
```

We can then run the following in the command line:

```
g++ --std=c++14 -Iinclude countlines.cpp -o countlines
./countlines workshop/shakespeare/*
```

and get the output:

```
shakespeare/allswellthatendswell = 4515
shakespeare/antonyandcleopatra = 5998
shakespeare/asyoulikeit = 4122
shakespeare/comedyoferrors = 2937
shakespeare/coriolanus = 5836
shakespeare/cymbeline = 5485
shakespeare/hamlet = 6045
shakespeare/juliuscaesar = 4107
shakespeare/kinglear = 5525
shakespeare/loveslabourslost = 4335
shakespeare/macbeth = 3876
```

```
shakespeare/measureforemeasure = 4337
shakespeare/merchantofvenice = 3883
shakespeare/merrywivesofwindsor = 4448
shakespeare/midsummersnightsdream = 3115
shakespeare/muchadoaboutnothing = 4063
shakespeare/othello = 5424
shakespeare/periclesprinceoftyre = 3871
shakespeare/README = 2
shakespeare/romeoandjuliet = 4766
shakespeare/tamingoftheshrew = 4148
shakespeare/tempest = 3399
shakespeare/timonofathens = 3973
shakespeare/titusandronicus = 3767
shakespeare/troilusandcressida = 5443
shakespeare/twelfthnight = 4017
shakespeare/twogentlemenofverona = 3605
shakespeare/winterstale = 4643
```

Reduction

This concept allows us to summarise the result of a map output. We

Lambda Functions

These are functions which can we use to map or reduce data without assigning the function a name or a space in memory. These are useful for functions that will only be used once for a specific application. A simple example of how we might use this is shown below with a sum function:

```
auto result = map( [](int x, int y){ return x + y;}, a, b );
```

The general format for these lambda functions to be passed to the `map()` or `reduce()` functions is

```
[]( arguments ){ function code; }
```

Map/Reduce

Parallel C++ Using IntelTBB

For Loops

The simplest element of Intel TBB is the `tbb::parallel_for` construct, it is used in the following format:

```
tbb::parallel_for( range, kernel );
```

where

- `range` is the set of values to iterate over
- `kernel` is a lambda function to be called for the subset of range values in the loop

An example of a kernel function would be

```
[&](tbb::blocked_range<int> r)
{
   for (int i=r.begin(); i<r.end(); ++i)
   {
       values[i] = std::sin(i * 0.001);
   }
}
```

so for each `i`, `sin(i*0.001)` is stored in the `i`th element of the vector `values`.

This works by TBB automatically assigning a subset of the range values of `i` to worker threads running on the computer cores.

Reducing

Another construct is the `tbb::parallel_reduce` - this allows us to parellelise loops that involve updating shared variables, as seen in the Reduce section previously. It is used as follows:

```
auto result = tbb::parallel_reduce( range, identity_value, kernel, reduction_function );
```

where

- `range` is used as in `tbb::parallel_for`
- `identity_value` is the starting identity value for the reduction (e.g. 1 for a multiplication)
- `reduction_function` is the function used for reducing the values
- `kernel` is used similarly as that in `tbb::parallel_for`

The difference in the `kernel` element is the signature, in `tbb::parallel_reduce` it takes the form

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

    return running_total;
}
```

We can see comparing it to the Functional Programming section, this has the capability for more than just a reduction, but is more similar to the map/reduce construct.

Going back to the `tbb:parallel_for` example, we can reduce the result using `tbb::parallel_reduce` while eliminating the need for the `tbb:parallel_for` altogether:

```
int main(int argc, char **argv)
{
    auto total = tbb::parallel_reduce(
                  tbb::blocked_range<int>(0,10000),
                  0.0,
```

```
                [](tbb::blocked_range<int> r, double running_total)
    {
        for (int i=r.begin(); i<r.end(); ++i)
        {
            running_total += std::sin(i * 0.001);
        }

        return running_total;
    }, std::plus<double>());

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

    return 0;
}
```

A Parallel Map/Reduce