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:

```cpp
#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 might usually do this in a loop, for example in the form of:

```
for (int result : results){
    total += result;
}
```

We can similarly create a reduce function as a complement to the `map` function:

```
template<class FUNC, class T>
T reduce(FUNC func, const std::vector<T> &values, const T &initial)
{
    if (values.empty())
    {
        return initial;
    }
    else
    {
        T result = initial;

        for (const T &value : values)
        {
            result = func(result, value);
        }

    return result;
    }
}
```

Example: Again we can use this for the shakespeare example to get a total count of the lines from all the plays included in the folder. We modify the `main` function to:

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

    auto results = map( count_lines, filenames );

    auto total = reduce( sum, results );

    std::cout << "The total number of lines is " << total << std::endl;

    return 0;
}
```

We then compile and run this to get the result:

```
The total number of lines is 119685
```

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:

```cpp
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

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

For example if we wanted to print a vector which has the squared elements of the input vector we could do as follows:

```cpp
#include "part1.h"

using namespace part1;

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

    auto squares = map( [](int x){ return x*x; }, a );

    print_vector(squares);

    return 0;
}
```

Compiling and running gives the result:

```
[ 1 4 9 16 25 ]
```

Note: we can use the same syntax to create a simple function as well.

Map/Reduce

This is the process of mapping two or more arrays and reducing the value to a single result. A general `mapReduce` function is given by:

```
template<class MAPFUNC, class REDFUNC, class... ARGS>
auto mapReduce(MAPFUNC mapfunc, REDFUNC redfunc, const std::vector<ARGS>&... args)
{
    typedef typename std::result_of<MAPFUNC(ARGS...)>::type RETURN_TYPE;

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

    if (nvals == 0)
    {
        return RETURN_TYPE();
    }

    RETURN_TYPE result = mapfunc(args[0]...);

    if (nvals == 1)
    {
        return result;
    }

    for (size_t i=1; i<nvals; ++i)
    {
        result = redfunc( result, mapfunc(args[i]...) );
    }

    return result;
}
```

Below is a simple example of how we might use this to create a multiplication map and sum reduction:

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

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

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 } );

    auto result = mapReduce( multiply, sum, a, b );

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

    return 0;
}
```

which gives output 130.

Example

This example shows how we can use all these concepts to write a program. We start with a program which calculates energy between two groups of point particles using a double loop:

```
#include "part1.h"

using namespace part1;

double calculate_energy(const Point &point1,
                        const Point &point2)
{
    return 1.0 / (0.1 + calc_distance(point1, point2));
}

double calculate_energy(const std::vector<Point> &group1,
                        const std::vector<Point> &group2)
{
    double total = 0;

    for (const Point &point1 : group1)
    {
        for (const Point &point2 : group2)
        {
            total += calculate_energy(point1, point2);
        }
    }

    return total;
}

int main(int argc, char **argv)
{
    auto group_a = create_random_points(5000);
    auto group_b = create_random_points(5000);

    auto energy = calculate_energy(group_a, group_b);

    std::cout << "Total energy = " << energy << std::endl;

    return 0;
}
```

We now want to replicate this using a map/reduce. We can replace the second `calculate_energy` function with the following `mapreduce_energy` function:

```
#include <functional>
double mapreduce_energy(const std::vector<Point> &group1,
                        const std::vector<Point> &group2)
{
    double total = 0;

    for (const Point &point1 : group1)
    {
        total += mapReduce( [=](const Point &point)
                            {
                                return calculate_energy(point, point1);
                            },
                            std::plus<double>(), group2 );
    }

    return total;
}
```

We can see here the inside loop has been replaced, although the outer loop remains. Hence we can also replace this loop to create a double map reduce:

```
double mapreduce_energy(const std::vector<Point> &group1,
                        const std::vector<Point> &group2)
{
    return mapReduce( [=](const Point &point1)
                      {
                          return mapReduce([=](const Point &point2)
                          {
                              return calculate_energy(point1, point2);
                          },
                          std::plus<double>(), group2 );
                      },
                      std::plus<double>(), group1 );
}
```

For loops v Map/Reduce

We might ask why use a map/reduce instead of a for loop- for loops are much easier to code and conceptually more understandable. The reason for this is that a map is inherently parallelisable as the order of calculations within the map and the reduce are irrelevant, however loops are inherently serial, often one iteration depends on the previous, e.g.

```
for (int i : vec)
    {
        total += vec;
    }
```

updates the `total` variable obtained in the previous loop.

It is possible to parallelise this but it is much more risky and likely to lead to bugs.

Parallel C++ Using IntelTBB

For Loops

The simplest element of Intel TBB is the `tbb::parallel_for` construc./countlines workshop/shakespeare/*t, 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

Here we try and recreate the multiplication/sum map/reduce that we did in the Map/Reduce section for serial programming, this time using TBB. For this we need to use `tbb::parallel_reduce` to create a new `mapReduce` function:

```
#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,
               const std::vector<int> &arg2)
{
    int nvals = std::min( arg1.size(), arg2.size() );

    return tbb::parallel_reduce(
              tbb::blocked_range<int>(0,nvals),
              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],arg2[i]) );
                  }
                  return running_total;
              }, redfunc );
}
```

We can then use this as before:

```
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 } );

    auto result = mapReduce( std::multiplies<int>(),
                             std::plus<int>(), a, b );

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

    return 0;
}
```

to again get the result of 130.