Part B - Languages

Threading Building Blocks

Introduce the TBB parallel template library
Describe the concept of splittable ranges
Show examples that implement common parallel patterns

Introduction | Map | Reduce | Scan | Pipeline | Fork-Join | Exercises

The Standard Template Library (STL) of C++ provides containers and algorithms for programmers to use in their applications. The STL implements parametric polymorphism using serial programming alone. Threading Building Blocks is a parallel complement to this library that offers a high-level, general purpose, feature-rich library for implementing parametric polymorphism using threads. It includes a variety of containers and algorithms that execute in parallel and has been designed to work without requiring any change to any compiler.

This chapter introduces Threading Building Blocks and the concept of a range. This chapter includes sample code that demonstrates how to access algorithms for map, reduce, scan, pipeline and fork-join problems.

INTRODUCTION

Threading Building Blocks (TBB) is a template library created by Intel that augments the C++ language with capabilities for task parallelism. It is an active open source project under the auspices of Intel. It is portable, composable and available for Windows, Linux and OS X operating systems. Its web site is at threadingbuildingblocks dot org. The release described here is 4.4 update 2.

TBB is available either as a stand-alone library or as part of the Parallel Studio XE Suite used in this course. To access TBB, you need to turn on specific settings:

- Intel C++ compiler: set Configuration Properties->Intel Performance Libraries->Use Intel TBB to Yes
- Visual Studio C++ compiler with the Intel Performance Libraries included: set Configuration Properties->Intel Preformance Libraries->Use Intel TBB to Yes
- Visual Studio C++ compiler without the Intel Performance Libraries included: set the Environment variables to point to the TBB installation and add them to the include and lib VC++ directories

Principal Features

Since TBB is a library and not a compiler extension, it does not implement vectorization. To add vectorization, we can use Cilk Plus, which is composable with TBB.

Namespaces

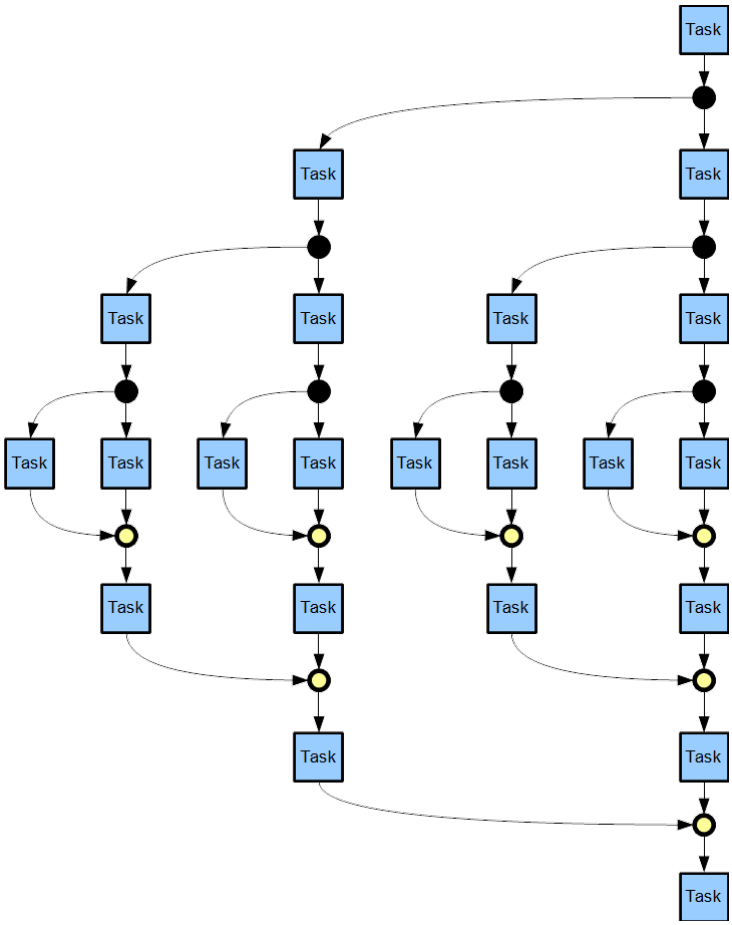
Public TBB identifiers are defined in the `tbb` and `tbb::flow` namespaces.

Documentation

The TBB documentation includes tips to help determine at which point parallel execution becomes worthwhile.

The Fork-Join Model

TBB implements a fork-join model of parallelism as illustrated in the figure below. It supports the nesting of tasks to any depth.



Fork-Join Control Flow Implementing a Recursive Divide and Conquer Algorithm

The Range Concept

TBB implements its fork-join model using the concept of a range. A TBB range is a collection that is recursively divisible into two parts over an iteration space. The programmer specifies the task that TBB is to execute on each base subrange.

A range has a splitting constructor that divides the range into nearly equal subranges. Typically, this splitting method yields the best parallelism.

TBB uses standard mathematical notation to describe a range. **`[0,n)`** denotes a half-open set; that is, a set from 0 to n in which 0 is included while n is excluded.

blocked_range

TBB defines a templated recursive range class for describing one, two and three-dimensional iteration spaces over type T. The one-dimensional template typically used with iteration spaces over integral types is **`blocked_range`**. The splitting constructor for a **`blocked_range`** object splits a range (or subrange) into two subranges (or subsubranges), replacing the original with the first subrange and returning the second subrange.

The following statements create an instance of a **`blocked_range`** named **`range0`** and then split that range into **`range1`** and **`range0`**:

```
// Construct a range from 0 inclusive to 40 exclusive [0,40)
blocked_range<int> range0(0 ,40);

// Split the range r0 into two subranges [0,20) and [20,40)
blocked_range<int> range1(range0);
```

Under the fork-join model, splitting continues until the iteration space has been subdivided into subranges of base size. The

number of iterations within a base subrange is called a *chunk*. The programmer specifies the task for TBB to execute on each chunk through a reference to the **blocked_range** object. The task specified by the programmer extracts the subrange of a chunk by calling the **begin()** and **end()** member functions on the referenced **blocked_range** object:

```
...(const blocked_range<int>& range) {
    for (auto i = range.begin(); i != range.end(); i++) {
        // programmer-specified task to be executed in serial
        // on this chunk
    }
}
```

The iterators returned by the **begin()** and **end()** member functions on the **blocked_range** object are random-access iterators. A random-access iterator has bi-directional as well as non-sequential functionality, similar to pointers. For more information on random-acess iterators and other types of iterators, see cplusplus.com.

Chunking

By default, TBB chooses the size of a chunk automatically. To make parallelization worthwhile, Intel suggests that the iterations within a chunk consume at least a million clock cycles. Otherwise, parallel overhead dominates.

TBB lets the programmer override the default and select the chunk size through partitioner and grainsize controls.

TBB supports three mechanisms:

- **auto_partitioner()** - turns on automatic chunking (default)
- **simple_partitioner()** - turns off automatic chunking
- **affinity_partitioner** - for successive parallelizations (no covered here)

TBB defines *grainsize* as the minimum threshold for parallelization. Intel suggests that each serial set of iterations on a chunk takes at least 100,000 clock cycles and recommends setting the grainsize to 100,000 and halving it each run until optimum performance is reached.

We set grainsize through the optional third argument when we instantiate a blocked range:

```
size_t grainsize = 100000;
blocked_range<int> r(0, size, grainsize);
```

Selecting **simple_partitioner()** with **grainsize** specified for a TBB algorithm (see below) ensures that the chunk size for any subrange will lie within the limits:

```
grainsize/2 <= chunk size <= grainsize
```

MAP

TBB implements the Map Pattern through its **parallel_for** template. This template recursively splits a range into subranges and applies a function to each subrange. A call to this template can take one of the following forms:

- **parallel_for(*first*, *last*, *function* [, partitioner]);**
- **parallel_for(*first*, *last*, *stride*, *function* [, partitioner]);**
- **parallel_for(*range*, *function* [, partitioner]);**

Documentation is available here.

The template declarations are:

```
template<typename Index, typename Func>
Func parallel_for( Index first, Index last, const Func& func [, partitioner]);

template<typename Index, typename Func>
```

```
Func parallel_for( Index first, Index last, Index stride, const Func& func
 [, partitioner]);

template<typename Range, typename Body>
void parallel_for( const Range&, const Body& body [, partitioner]);
```

The following examples use anonymous functions (lambda expressions) to define the map operations. The serial version is on the left. The equivalent parallel version is on the right.

```
// TBB - serial
// tbb_map_serial.cpp

#include <iostream>


int main() {
    int a[] = {1,2,3,4,5,6,7,8};
    int b[] = {1,1,1,1,1,1,1,1};
    auto iaxpy = [&](int i) {
        b[i] = 2 * a[i] + b[i];
    };
    for (int i = 0; i < 8; i++)
     iaxpy(i);
    for (int i = 0; i < 8; i++)
        std::cout << b[i] << ' ';
    std::cout << std::endl;
}
```

```
// TBB - parallel_for - simple range
// tbb_map.cpp

#include <iostream>
#include <tbb/tbb.h>

int main() {
    int a[] = {1,2,3,4,5,6,7,8};
    int b[] = {1,1,1,1,1,1,1,1};
    auto iaxpy = [&](int i) {
        b[i] = 2 * a[i] + b[i];
    };

    tbb::parallel_for(0, 8, iaxpy);
    for (int i = 0; i < 8; i++)
        std::cout << b[i] << ' ';
    std::cout << std::endl;
}
```

```
// TBB - serial
// tbb_map_serial.cpp

#include <iostream>


int main() {
    int a[] = {1,2,3,4,5,6,7,8};
    int b[] = {1,1,1,1,1,1,1,1};
    auto iaxpy = [&](
     int i) {
        b[i] = 2 * a[i] + b[i];
    };



    for (int i = 0; i < 8; i++)
     iaxpy(i);
    for (int i = 0; i < 8; i++)
        std::cout << b[i] << ' ';
    std::cout << std::endl;
}
```

```
// TBB - parallel_for - with tiling
// tbb_map_tiling.cpp

#include <iostream>
#include <tbb/tbb.h>

int main() {
    int a[] = {1,2,3,4,5,6,7,8};
    int b[] = {1,1,1,1,1,1,1,1};
    auto iaxpy = [&](
     tbb::blocked_range<int>& r) {
        for (int i = r.begin();
         i != r.end(); i++)
            b[i] = 2 * a[i] + b[i];
    };
    tbb::blocked_range<int> range(0,8);
    tbb::parallel_for(range, iaxpy);
    for (int i = 0; i < 8; i++)
        std::cout << b[i] << ' ';
    std::cout << std::endl;
}
```

Note that the **for** loop defined by the lambda expression executes in serial within the blocked range for each chunk.

REDUCE

TBB implements the Reduce Pattern through its **parallel_reduce** template. This template performs a parallel reduction on a specified range by recursively splitting that range into subranges, applying a function to each base subrange and applying a reduction operation on the reduced result returned by the function. The reductions performed by the function on each base subrange is a serial operation.

A call to this template can take one of the following forms:

- `parallel_reduce(range, identity, function, reduction [, partitioner]);`
- `parallel_reduce(range, body [, partitioner]);`

Documentation for this template is available here.

The template declaration are:

```
template<typename Range, typename Value, typename Func, typename Reduce>
Value parallel_reduce( const Range&, const Value&, const Func&,
 const Reduce& [, partitioner]);

template<typename Range, typename Body>
void parallel_reduce( const Range&, Body& [, partitioner]);
```

The following example uses

- a lambda expression for the reduction operation (`add`)
- a lambda expression for the function that operates on each chunk of the range

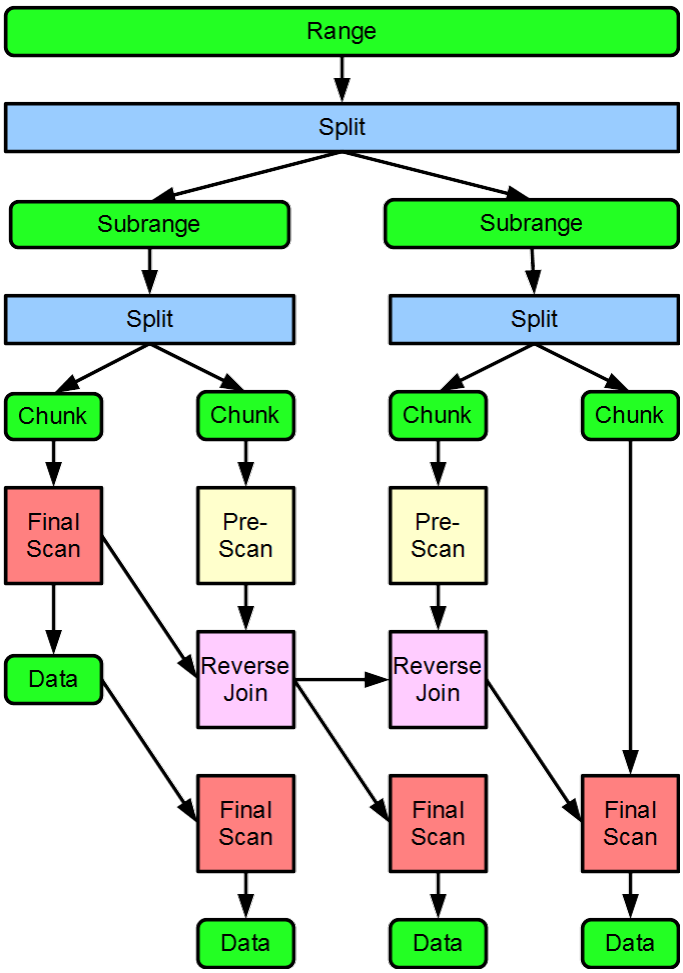
```
// TBB - parallel_reduce - blocked range
// tbb_reduce.cpp

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

template<typename T, typename C>
T reduce(
    const T* first, // start of range
    const T* last,  // end of range
    T identity,     // initial value
    C combine       // combination expression
    )
{
    return
     tbb::parallel_reduce(
         tbb::blocked_range<const T*>(first, last),
         identity,
         // operates on each chunk of the range
         [&](tbb::blocked_range<const T*> r, T partial) -> T {
             for (auto x = r.begin(); x != r.end(); x++)
                 partial = combine(*x, partial);
             return partial;
         },
         // operates on the result on each chunk's reduction
         combine
     );
}

int main() {
    int a[] = { 1, 2, 3, 4, 5, 6, 7, 8 };
    auto add = [](int a, int b) { return a + b; };
    int sum = reduce(a, a + 8, 0, add);
    std::cout << "Sum = " << sum << std::endl;                    Sum = 36
}
```

The characters between the function's closing parenthesis and its opening brace (`-> T`) identify the return type from this anonymous function.

SCAN

TBB implements the Scan Pattern through its **`parallel_scan`** template. This template performs an inclusive parallel prefix-scan across a specified range. The first argument identifies the range and the second argument identifies the functor (function object) that performs the upsweep (reduction) and downsweep (prefix-scan) operations on each base subrange.

Documentation for this template is available here. The algorithm that implements this pattern is a variation on the balanced tree algorithm. The operations that constitute the algorithm are fully described in this documentation.



Pre-Scan (Upsweep Reduction), Reverse-Join and Final-Scan (Downsweep) Algorithm

A call to this template takes the following form:

- **`parallel_scan(`*`range`*`, `*`body`*` [, partitioner]);`**

The template declaration is:

```
template<typename Range, typename Body>
void parallel_scan( const Range&, Body& [, partitioner&]);
```

The **`Body`** functor defines the operation performed in the algorithm. For this functor, we define the following member functions:

- **`Body::Body(...)`** - constructor that receives data that the member functions require
- **`void Body::operator()(const Range& subrange, tbb::pre_scan_tag)`** - accumulates on **`subrange`** (reduction)
- **`void Body::operator()(const Range& subrange, tbb::is_final_scan)`** - computes scan result on **`subrange`** and updates (downsweep)
- **`Body::Body(Body& b, tbb::split)`** - splits object **`b`** into subobjects
- **`void Body::reverse_join(Body& b)`** - merges the accumulated value of object **`b`** into the current object

- **`void Body::assign(Body& b)`** - assigns the accumulator of object **`b`** to the current object

The following example uses a lambda expression to define the reduction operation and implements the call operators for the functor in a single function definition:

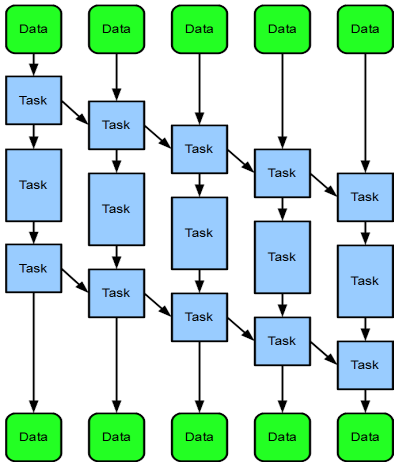
```
// TBB - parallel_scan - blocked range
// tbb_6.cpp

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

template<typename T, typename C>
class Body {
    T accumul;
    T* const out;
    const T* const in;
    const T identity;
    const C combine;
public:
    Body(T* out_, const T* in_, T i, C c) :
     accumul(i), out(out_), in(in_),
     identity(i), combine(c) {}
    T get_accumul() const { return accumul; }
    template<typename Tag>
    void operator()(
     const tbb::blocked_range<int>& r, Tag) {
        T temp = accumul;
        for (int i = r.begin(); i != r.end();
         i++) {
            temp = combine(temp, in[i]);
            if (Tag::is_final_scan())
                out[i] = temp;
        }
        accumul = temp;
    }
    Body(Body& b, tbb::split) :
     accumul(b.identity), out(b.out), in(b.in),
     identity(b.identity), combine(b.combine) {}
    void reverse_join(Body& a) {
     accumul = combine(accumul, a.accumul); }
    void assign(Body& b) { accumul = b.accumul; }
};

template<typename T, typename C>
T scan(
    T* out,
    const T* in,
    int n,
    T identity,
    C combine
    )
{
    Body<T, C> body(out, in, identity, combine);
    tbb::parallel_scan(
     tbb::blocked_range<int>(0, n), body);
    return body.get_accumul();
}

int main() {
    int a[] = { 1, 2, 3, 4, 5, 6, 7, 8 };
    int b[8];
    auto add = [](int a, int b) { return a + b; };
    int sum = scan(b, a, 8, 0, add);
```

```
    for (int i = 0; i < 8; i++)
        std::cout << b[i] << " ";
    std::cout << std::endl;                          1 3 6 10 15 21 28 36
    std::cout << "Sum = " << sum << std::endl;       Sum = 36
}
```

tbb::split distinguishes the splitting constructor from the copy constructor. **Tag** is **tbb::pre_scan_tag** on the reduction pass and **tbb::final_scan_tag** on the downsweep pass.

PIPELINE

A pipeline is a linear sequence of stages with data flowing through it and each stage transforming the data. Partitioning the data into subsets allows the pattern to complete processing of one subset before starting to process some other subsets. The pipeline pattern is illustrated in the Figure below. Note the absence of dependencies between the intermediate tasks.



A Pipeline Pattern

Consider the following serial implementation of a pipeline:

```
// TBB - parallel_pipeline
// tbb_serial_pipeline.cpp

#include <iostream>

template<typename T, typename U, typename V, typename W>
void sps_pipeline(
    U begin,
    V middle,
    W end
    )
{
    while (T t = begin()) {
        T u = middle(t);
        end(u);
    }
}

int main() {                                                    7
    int n = 8;                                                  6
    auto begin = [&]() { return n--; };                         5
    auto middle = [](int a) { return a - 1; };                  4
    auto end = [](int a) { std::cout << a << '\n'; };           3
    sps_pipeline<int, decltype(begin), decltype(middle),        2
```

```
     decltype(end)>(begin, middle, end);                                    1
}                                                                           0
```

The **parallel_pipeline** function manages the execution of a pipeline through a set of functions that operate on each item in a set. Each function maps an object of one type to another using the call operator. The types form a chain that describes the sequence of stages in the pipeline. The first parameter receives the maximum number of token in the chain. The second parameter receives the chain itself. The chain consists of tokens concatenated using the **&** operator. This operator requires that the output type of one stage matches the input type of the following stage. The first function in the chain is special in that it checks if the input stream is finished and if so, stops the process returning a null value.

Documentation for this template is available here.

The prototype for the **parallel_pipeline** function is:

```
void parallel_pipeline( size_t, const filter_t&);
```

and the template declarations for the **filter_t** class are:

```
template<typename T, typename U> class filter_t;

template<typename T, typename U, typename Func>
filter_t<T, U> make_filter(filter::mode, const Func&);

template<typename T, typename V, typename Func>
filter_t<T, U> operator&(const filter_t<T, V>&,
 const filter_t<V, U>&);
```

These templates are used in the following example to create a filter chain. Lambda expressions are used to define the initial serial, parallel and final serial stages of the pipeline:

```
// TBB - parallel_pipeline
// tbb_parallel_pipeline.cpp

#include <iostream>
#include <tbb/tbb.h>
#include <tbb/pipeline.h>

template<typename T, typename U, typename V, typename W>
void sps_pipeline(
    int ntokens,
    U serial_begin,
    V parallel,
    W serial_end
    )
{
    tbb::parallel_pipeline(ntokens,
        tbb::make_filter<void, T>(
         tbb::filter::serial_in_order,
         [&](tbb::flow_control& fc) -> T {
            T item = serial_begin();
            if (!item) fc.stop();
            return item;
         }) &
        tbb::make_filter<T, T>(
         tbb::filter::parallel, parallel) &
        tbb::make_filter<T, void>(
         tbb::filter::serial_in_order, serial_end)
    );
}

int main() {                                                                7
    int n = 8;                                                              6
    auto begin = [&]() { return n--; };                                     5
```

```
    auto middle = [](int a) { return a - 1; };                          4
    auto end = [](int a) { std::cout << a << '\n'; };                   3
    sps_pipeline<int, decltype(begin), decltype(middle),                2
     decltype(end)>(3, begin, middle, end);                             1
}                                                                       0
```

The pipeline pattern has weaker synchronization requirements than the map pattern.

WORKPILE

The workpile pattern is an extension of the map pattern that

FORK-JOIN

The fork-join pattern implements task parallelism. TBB supports a simple form and a comprehensive form:

- list of tasks
- control of spawns and synchronizations

Invoke

The **`tbb::parallel_invoke()`** templated function evaluates up to 10 function objects in parallel. Each functor argument to this function should define a **`void operator()() const`** member function that takes no arguments. Lambda expressions may serve as alternatives. Control transfers from this function after it has executed the specified tasks and rejoined the forked tasks.

```
tbb::parallel_invoke(
 [=]foo(i);,
 [=]goo(i);,
 [=]hoo(i);
)
```

Task Group

The **`tbb::task_group`** class supports explicit forking and join operations. An explicit call to **`wait()`** is necessary before the **`tbb::task_group`** object goes out of scope.

```
tbb::task_group group;
i++;
group.run([=,&a] foo(a[i]);); // forks task foo
i++;
group.run([=,&a] goo(a[i]);); // forks task goo
i++;
group.run([=,&a] hoo(a[i]);); // forks task hoo
i++;
group.wait(); // waits for all forked tasks to complete
```

`i` is captured by value since the tasks might execute after the incrementations. **`a`** is captured by reference so that it can be updated during the actual execution of the lambda expression.

EXERCISES

- Threading Building Blocks Home Page
- Intel Threading Building Blocks Developer Reference
- Intel TBB Design Patterns
- Design Patterns for Parallel Computing