High-Performance Computer Architectures
Practical Course

Thread Building Blocks (TBB)

Prof. Dr. Ivan Kisel
Robin Lakos
Akhil Mithran
Oddharak Tyagi

June 11, 2025

Problems with Explicit Parallelization

Gaining performance from multi-core requires parallel programming
Even a simple “parallel for” is tricky for a non-expert to write well with explicit threads

Two aspects need to be considered:

- Correctness: Avoiding race conditions and deadlock
- Performance: Efficient use of resources
  - Hardware threads (match parallelism to hardware threads)
  - Memory space (choose right evaluation order)
  - Memory bandwidth (reuse cache)

Key Features of Intel Threading Building Blocks

You specify task patterns instead of threads (focus on the work, not the workers)

- Library maps user-defined logical tasks onto physical threads, efficiently using cache and balancing load
- Full support for nested parallelism, targets threading for robust performance
- Designed to provide portable scale-able performance for computationally intense portions of shrink-wrapped applications. Compatible with other threading packages
- Designed for CPU bound computation, not I/O bound or real-time.
- Library can be used in concert with other threading packages such as OpenMP.

Example: parallel_for

We want to apply Foo to each element of an array, and it is safe to process each element concurrently. The sequential code to do this is:

```
void SerialApplyFoo( float a[], size_t n ) {
    for( size_t i=0; i≠n; ++i )
        Foo(a[i]);
}
```

The template function `tbb::parallel_for` breaks this iteration space into chunks, and runs each chunk on a separate thread.

Example: parallel_for

First we have to convert the loop body into a form that operates on a chunk. The form is an STL-style function object, called the body object, in which `operator()` processes a chunk. The following code declares the body object.

```
#include "tbb/blocked_range.h"
class ApplyFoo {
    float *const my_a;
public:
    void operator()( const blocked_range<size_t>& r ) const
    {
        float *a = my_a;
        for( size_t i=r.begin(); i!=r.end(); ++i )
            Foo(a[i]);
    }
    ApplyFoo( float a[] ) : my_a(a) {}
};
```

Once you have the loop body written as a body object, use the template function `parallel_for`, as follows:

```
#include "tbb/parallel_for.h"
void ParallelApplyFoo( float a[], size_t n ) {
    parallel_for(blocked_range<size_t>(0,n), ApplyFoo(a));
}
```

Requirement for parallel_for body object

- Copy constructor

```
Body::Body(const Body&)
```

- Destructor

```
Body::~Body()
```

- In most cases, the implicitly generated copy constructor and destructor work correctly
- Apply body to subrange

```
void Body::operator()(Range& subrange) const
```

- `operator()` should not modify the body. Otherwise the modification might or might not become visible to the thread that invoked `parallel_for`, depending upon whether `operator()` is acting on the original or a copy.
- Therefore, `parallel_for` requires that the body object's `operator()` be declared as `const`

Example: parallel_reduce

Consider the following loop doing reduction

```
float SerialSumFoo( float a[], size_t n ) {
    float sum = 0;
    for( size_t i=0; i!=n; ++i )
        sum += Foo(a[i]);
    return sum;
}
```

If iterations are independent, `parallel_reduce` can be used to parallelize this loop as follows:

```
float ParallelSumFoo( const float a[], size_t n ) {
    SumFoo sf(a);
    parallel_reduce( blocked_range<size_t>(0,n), sf );
    return sf.my_sum;
}
```

where the class `SumFoo` specifies details of the reduction, such as how to accumulate sub-sums and combine them.

Example: parallel_reduce

```
class SumFoo {
    float* my_a;
public:
    float my_sum;
    void operator()( const blocked_range<size_t>& r ) {
        float *a = my_a;
        float sum = my_sum;
        for( size_t i=r.begin(); i!=r.end(); ++i )
            sum += Foo(a[i]);
        my_sum = sum;
    }

    SumFoo( SumFoo& x, split ) : my_a(x.my_a), my_sum(0) {} //splitting constructor

    void join( const SumFoo& y ) {my_sum+=y.my_sum;} // to accumulate results of subtasks

    SumFoo(float a[] ) : my_a(a), my_sum(0) {}
};
```

Note:

- `operator()` is not `const`. This is because it must update `SumFoo::my_sum`.
- `SumFoo` has a splitting constructor and a method join that must be present for `parallel_reduce` to work. The splitting constructor takes as arguments a reference to the original object, and a dummy argument of type split, which is defined by the library. The dummy argument distinguishes the splitting constructor from a copy constructor.