Threading Build Block

* Intel® Threading Building Blocks enables you to specify *logical paralleism* instead of threads.
* Intel® Threading Building Blocks targets *threading for performance*.
* Intel® Threading Building Blocks is *compatible* with other threading packages.
* Intel® Threading Building Blocks emphasizes *scalable, data parallel programming*.
* Intel® Threading Building Blocks relies on *generic programming*.

## Parallel For

loop body written as a body object

**#include "tbb/tbb.h**"  
   
**using namespace tbb;**  
   
**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)**  
 **{}**  
**};**

then

void ParallelApplyFoo( float a[], size\_t n ) {  
 parallel\_for(blocked\_range<size\_t>(0,n), ApplyFoo(a));  
}

But do we need do it everytime?

With Lambda

void ParallelApplyFoo( float\* a, size\_t n ) {  
 parallel\_for( blocked\_range<size\_t>(0,n),   
 **[=](const blocked\_range<size\_t>& r) {**  
 **for(size\_t i=r.begin(); i!=r.end(); ++i)**   
 **Foo(a[i]);**   
 **}**  
 );  
}

The [=] specifies that capture is by value. Writing [&] instead would capture the values by reference.

#include "tbb/tbb.h"  
   
using namespace tbb;  
   
#pragma warning(disable: 588)  
   
void ParallelApplyFoo(float a[], size\_t n) {  
 parallel\_for(size\_t(0), n, [=](size\_t i) {Foo(a[i]);});  
}

### CAUTION

Typically a loop needs to take at least a million clock cycles to make it worth usingparallel\_for. For example, a loop that takes at least 500 microseconds on a 2 GHz processor might benefit from parallel\_for.

The default automatic chunking is recommended for most uses.

#include "tbb/tbb.h"  
   
void ParallelApplyFoo( float a[], size\_t n ) {  
 parallel\_for(blocked\_range<size\_t>(0,n**,G**), ApplyFoo(a),   
 simple\_partitioner());  
}

A rule of thumb is that grainsize iterations of operator() should take at least 100,000 clock cycles to execute.

A general rule of thumb for parallelizing loop nests is to parallelize the outermost one possible.

Usingaffinity\_partitioner can significantly improve performance when:

* The computation does a few operations per data access.
* The data acted upon by the loop fits in cache.
* The loop, or a similar loop, is re-executed over the same data.
* There are more than two hardware threads available. If only two threads are available, the default scheduling in Intel® Threading Building Blocks (Intel® TBB) usually provides sufficient cache affinity

Henceaffinity\_partitioner should be considered a tool, not a cure-all, when there is a low ratio of computations to memory accesses.

|  |  |  |
| --- | --- | --- |
| artitioner | Description | When Used with blocked\_range(i,j,g) |
| simple\_partitioner | Chunksize bounded by grain size. | g/2 ≤ chunksize ≤ g |
| auto\_partitioner (default)[[4]](http://software.intel.com/sites/products/documentation/doclib/tbb_sa/help/tbb_userguide/Partitioner_Summary.htm#ftn4) | Automatic chunk size. | g/2 ≤ chunksize |
| affinity\_partitioner | Automatic chunk size and cache affinity. |  |

An auto\_partitioner is used when no partitioner is specified. In general, the auto\_partitioner or affinity\_partitioner should be used, because these tailor the number of chunks based on available execution resources.

float ParallelSumFoo( const float a[], size\_t n ) {  
 SumFoo sf(a);  
 parallel\_reduce( blocked\_range<size\_t>(0,n), sf );  
 return **sf.my\_**sum;  
}

The class SumFoo specifies details of the reduction, such as how to accumulate subsums and combine them. Here is the definition of class SumFoo:

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;  
 size\_t end = r.end();  
 for( size\_t i=r.begin(); i!=end; ++i )   
 sum += Foo(a[i]);   
 **my\_sum = sum;**   
 **}**  
   
 SumFoo( SumFoo& x, split ) : my\_a(x.my\_a), my\_sum(0) {}  
   
 void join( const SumFoo& y ) {my\_sum+=y.my\_sum;}  
   
 SumFoo(float a[] ) :  
 my\_a(a), my\_sum(0)  
 {}  
};

parallel\_reduce generalizes to any associative operation. In general, the splitting constructor does two things:

* Copy read-only information necessary to run the loop body.
* Initialize the reduction variable(s) to the identity element of the operation(s).

class SumFoo {  
 ...  
public:  
 float my\_sum;   
 void operator()( const blocked\_range<size\_t>& r ) {  
 ...  
 float **sum = 0;** // WRONG – should be 'sum = **my\_sum**".  
 ...  
 for( ... )   
 sum += Foo(a[i]);   
 my\_sum = sum;   
 }  
 ...  
};

This is because the SumFoo may be reused, especially when there is only one worker thread available.