Introducing Intel Tbb

Marco Corvo

CNRS and INFN

II SuperB Collaboration Meeting

Frascati December 13-16, 2011

What is Intel Tbb

1. Intel Threading Building Blocks is a runtime-based parallel programming model for C++ code that uses threads
2. It consists of a template-based runtime library to help you harness the latent performance of multicore processors
3. Tbb allows the user to write scalable applications that
   - Specify logical parallel structure instead of threads
   - Emphasize data parallel programming
   - Take advantage of concurrent collections and parallel algorithms

Where to find Tbb

- Tbb is available at http://threadingbuildingblocks.org
- it's available also as a commercial version
- Besides the documentation, there's a useful blog where developers can discuss with Intel gurus http://software.intel.com/en-us/blogs/category/intel-threading-building-blocks/

Disclaimer

- I think that one of the main source of confusion about parallelization rises from a
- Parallelize a given program or software requires much more than simply add some **pragma** statement to the source code
- **pragma**s are useful to unroll *for* loops, but when it comes to real parallelization and concurrency you **must** redesign your code

A simple example: Tbb parallel_for

- Suppose we want to apply a given function to each element of an array

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

- The first step in parallelizing this loop is to convert the loop body into a form that operates as required by `parallel_for`

```
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){}
};
```

A simple example: Tbb parallel_for II

- Now to execute the for loop in parallel

```
void ParallelApplyFoo( float a[], size_t n ) {
    parallel_for(blocked_range<size_t>(0,n), ApplyFoo
        (a));
}
```

- As you can see the structure becomes trickier than one could expect
- And this is the simplest thing you can do with Tbb

Concurrency

- Tbb provides highly concurrent container classes
- A concurrent container allows multiple threads to concurrently access and update items in the container
  - Typical STL libraries don't allow concurrency, unless you wrap them with a mutex, though reducing parallel speedup
- Example containers: *concurrent_vector*, *concurrent_hash_map*, *concurrent_queue*

Tbb Tasks

- Tasks are more specialized objects than `parallel_loop`
- If you design your software slicing the computation in elementary operations (tasks), Tbb task scheduler can decide the task size, number of threads to use and their schedule
- Simple example: a Fibonacci series

```
long SerialFibo(long j) {
  if (j<2) {
      return j;
  } else {
      return SerialFibo(j-2) + SerialFibo(j-1);
  }
}
```

Tbb Tasks II

In terms of Tbb tasks, the function is much different. First of all it's no more a function, but a class

```
class FibTask: public tbb::task {
public:
  const long n;
  long* const sum;
  FibTask(long _n, long* _s): n(_n),sum(_s){}
  task* execute() {
      if (n < CutOff) {
          *sum = SerialFibo(n);
      } else {
          long x, y;
          FibTask& a = *new(task::allocate_child())
              FibTask(n - 1, &x);
          FibTask& b = *new(task::allocate_child())
              FibTask(n - 2, &y);
          set_ref_count(3);
          spawn(b);
          spawn_and_wait_for_all(a);
          *sum = x + y;
      }
      return NULL;
  }
```

Tbb Tasks II

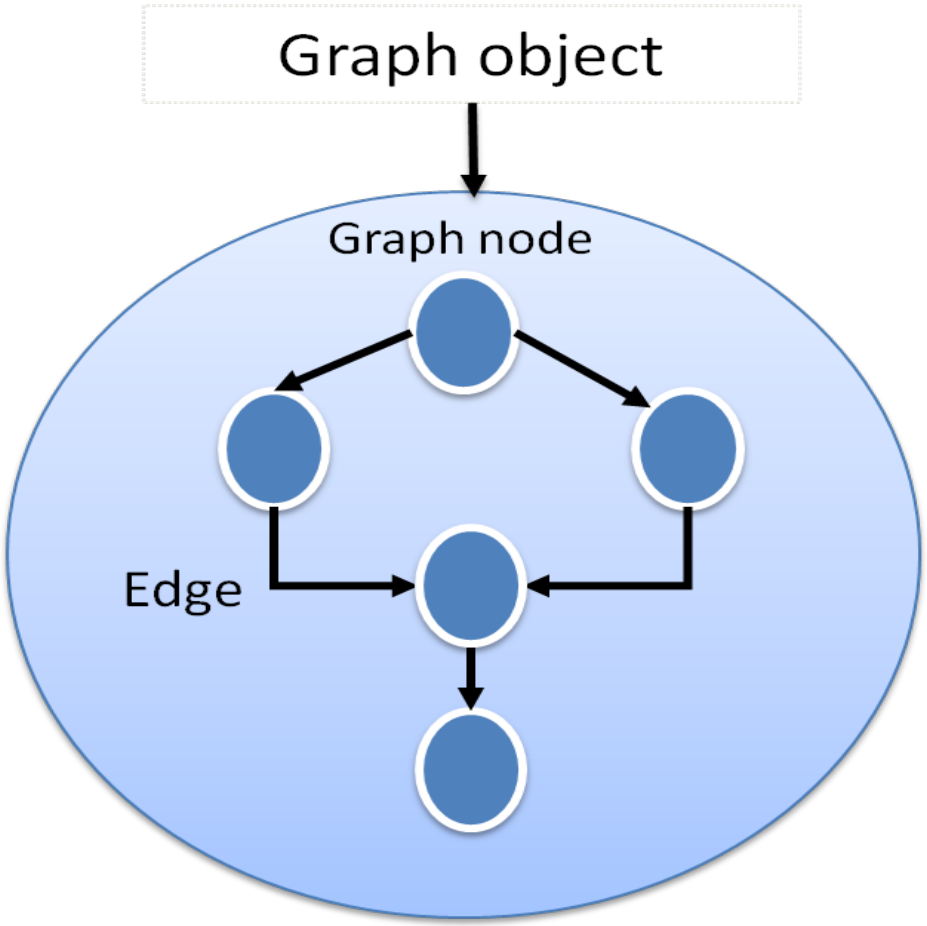
To call the Fibonacci 'function'

```
long ParallelFib(long j) {
  long sum;
  tbb::tick_count mainStartTime = tbb::tick_count::now();
  FibTask& a = *new(tbb::task::allocate_root()) FibTask(j
      , &sum);
  tbb::task::spawn_root_and_wait(a);
  tbb::tick_count mainStopTime = tbb::tick_count::now();
  double etime = (mainStopTime - mainStartTime).seconds()
      ;
  cout <<                                  << etime << endl;
  return sum;
}
```

Watch out!

As already said, a simple recursive function can take a much more complex shape when designed for parallelization.

The *flow_graph* environment I



- What caught our attention on Tbb was the `flow_graph` environment which is available since the last version of Tbb (4.0)
- `flow_graph` provides flexible and convenient API for expressing static and dynamic dependencies between computations
  - In our case we'd like to express dependencies among modules that, in the current framework, are executed sequencially

The *flow_graph* environment II

- *flow_graph* offers many different kind of nodes:
  - functional, that is they perform a user-provided computation
  - buffer, that is they keep a set of messages which are dispatched in an arbitrary order
  - queue, that is they dispatch messages to other nodes in a FIFO order
  - join, which collect messages from other nodes
- A couple of exercises as proof-of-concept has been done:
  1. How we can exploit Tbb in our quest for parallelism in the framework
  2. A quantitative measure of the potential speedup when parallelizing an event generator

Conclusions

- Tbb looks promising, but other options are available
- the point is that at some point we should find our way home...
- These months have been useful to dig into the old baBar framework and exercise with Tbb
- But again we have to put some sticks around and start with the real work