tbb::parallel_for

```
template<typename TI, typename TF>
void tbb::parallel_for( TI begin, TI end, const TF& f);
```

– Implements a parallel for loop for values in the range [begin,end)
– Each iteration ***may*** execute in parallel
  - There are no guarantees
– Iterations may be executed in any order
  - Entirely legal to do the iterations in opposite order to normal

– Programmer’s job: make sure all iterations are independent
– TBB run-time’s job: try to execute iterations as fast as possible

Implementing `tbb::parallel_for`

- How could we implement parallel_for?
  – Simple helper class for describing a range
- Start with a sequential version

```
template<class TI,class TF>
void parallel_for(TI beg,TI end, const TF &f)
{
   for(TI i=beg;i<end;i++){
      f( i );
   }
}
```

Implementing `tbb::parallel_for`

- How could we implement parallel_for?
  – Simple helper class for describing a range
- Start with a sequential version
- What about a Cilk version?
  – Q : What does the critical path look like?

```
template<class TI,class TF>
void parallel_for(TI begin, TI end, const TF &f)
{
   for(TI i=begin;i<end;i++){
     spawn f( i );
   }
}
```

Implementing `tbb::parallel_for`

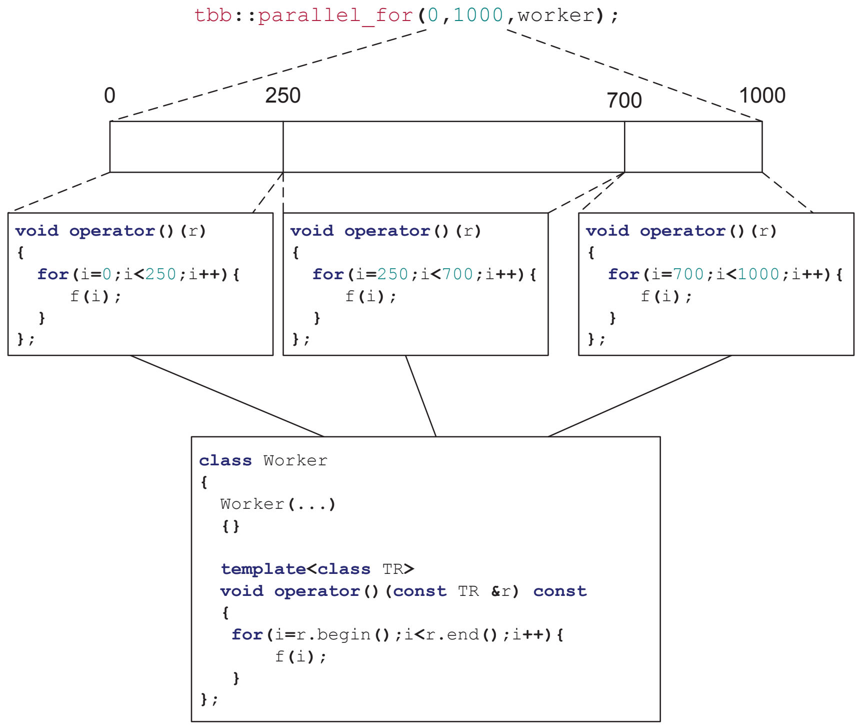
- How could we implement parallel_for?
  – Simple helper class for describing a range
- Start with a sequential version
- What about a Cilk version?
  – Let’s try again

```
template<class TI,class TF>
void parallel_for(TI begin, TI end, const TF &f)
{
   if(begin+1==end){
      spawn f( begin );
   }else{
      spawn parallel_for(begin, (begin+end)/2, f);
      spawn parallel_for((begin+end)/2, end, f);
   }
}
```

Implementing `tbb::parallel_for`

- This looks good, except what about *cost of work!*
- TBB will automatically apply **agglomeration**
  - Split range up into larger contiguous ranges
  - Hand ranges to function for processing in sequential loop
- How does it know what THRESH should be?
  - Dynamically varies based on time for last batch – auto-tune!

```
template<class TI,class TF>
void parallel_for(TI begin, TI end, const TF &f)
{
   if(end-begin < THRESH){
       for(TI i=begin;i<end;i++)
           spawn f( i );
   }else{
       spawn parallel_for(begin,  (begin+end)/2, f);
       spawn parallel_for((begin+end)/2, end, f);
   }
}
```



```
tbb::parallel_for(0,1000,worker);
```

```
void operator()(r)
{
  for(i=0;i<250;i++){
     f(i);
  }
};
```

```
void operator()(r)
{
  for(i=250;i<700;i++){
     f(i);
  }
};
```

```
void operator()(r)
{
  for(i=700;i<1000;i++){
     f(i);
  }
};
```

```
class Worker
{
  Worker(...)
  {}

  template<class TR>
  void operator()(const TR &r) const
  {
   for(i=r.begin();i<r.end();i++){
       f(i);
   }
};
```

References vs pointers

- C++ has references as well as pointers
  - Pointers use asterisk (*), references use ampersand (&)
- Some differences between pointers and references
  - References are guaranteed to be non-null
  - A reference always points at the same object
- Operations on the references happen to the original object

```
void MyPtrFunc(my_class *x)
{
   x->wibble();
   my_class *p=x;
   *x = my_class(5);
   my_class tmp;
   x=tmp;
}

my_class x;
MyPtrFunc(&x); // pass by pointer
```

```
void MyRefFunc(my_class &x)
{
   x.wibble();
   my_class *p=&x;
   x = my_class(5);
   my_class tmp;
   ~~&x=&tmp;~~
}

my_class x;
MyRefFunc(x); // pass by ref
```

Const methods

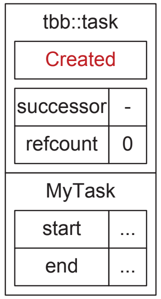
- Methods with the `const` modifier cannot change the object
  - Cannot modify the internal state of the object
  - Object is “the same” after any const method is called
- References with the `const` modifier cannot be changed
  - Can only call const methods and read object properties
  - Only methods marked const are considered const methods
    - Methods which don’t change state are “non-const” unless marked
- Const references are very useful
  - Pass by reference makes it cheap to pass object to function
  - Const-ness means object can be safely used in parallel

Scheduling through reference counts

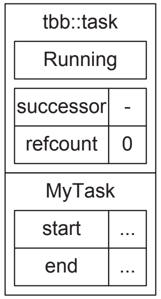
- Each task has a reference count and a successor task
- The reference count identifies whether a task is blocked
  - If the reference count is zero then the task could be run
  - But only if it has been given to the task scheduler
  - Legal to create a task and not give it to the scheduler
  - *Note the difference: “reference count” vs “C++ reference”*
- The successor task identifies the task blocked by this task
  - Same concept as “parent” in Cilk, but slightly more general
  - When a task completes it decrements the count of its successor

```
cilk void MyTask(int start, int end)
{
   if(cond())
       return 0;
   spawn MyTask(start,(start+end)/2);
   spawn MyTask((start+end)/2,end);
   DoSomethingFirst();
   sync;
   DoSomethingElse();
   return 0;
}
```

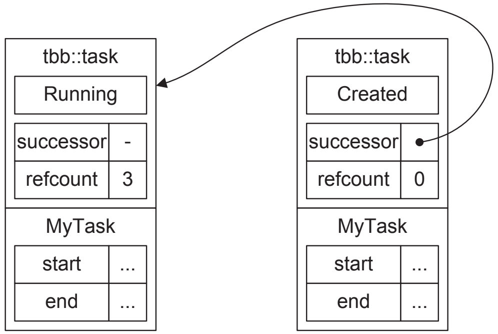
```
class MyTask
  : public tbb::task
{
   int start, end;

   MyTask(int _start, int _end)
   { start=_start;  end=_end; }

   tbb::task * execute()
   {
       if(cond())
           return 0;
       set_ref_count(3);
       MyTask &t1=*new(allocate_child()) MyTask(start,(start+end)/2);
       spawn(t1);
       MyTask &t2=*new(allocate_child()) MyTask((start+end)/2, end);
       spawn(t2);
       DoSomethingFirst();
       wait_for_all();
       DoSomethingElse();
   }
};

void CreateTasks(int start, int end)
{
   MyTask &root=*new(allocate_root()) MyTask(start,end);
   tbb::task::spawn_root_and_wait();
}
```



```
void CreateTasks(int start, int end)
{
   MyTask &root=*new(allocate_root()) MyTask(start,end);
   tbb::task::spawn_root_and_wait();
}
```



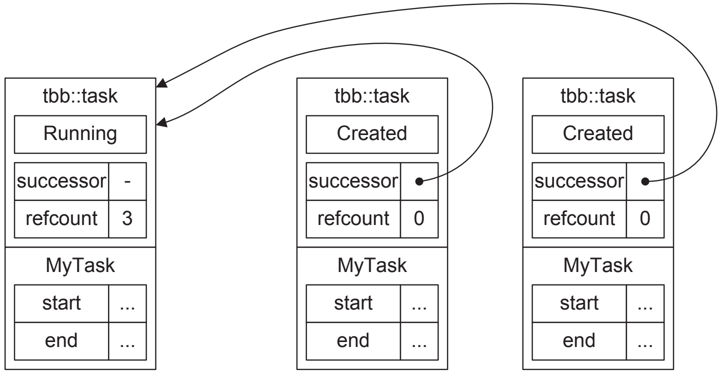
```
tbb::task * MyTask::execute()
{
  if(cond())
    return 0;
  set_ref_count(3);
  MyTask &t1=*new(allocate_child()) MyTask(start,(start+end)/2);
  MyTask &t2=*new(allocate_child()) MyTask((start+end)/2, end);
  spawn(t1);
  spawn(t2);
  DoSomethingFirst();
  wait_for_all();
  DoSomethingElse();
  return 0;
}
```



tbb::task
Running
successor -
refcount 3
MyTask
start ...
end ...

tbb::task
Created
successor
refcount 0
MyTask
start ...
end ...

```
tbb::task * MyTask::execute()
{
  if(cond())
    return 0;
  set_ref_count(3);
  MyTask &t1=*new(allocate_child()) MyTask(start,(start+end)/2);
  MyTask &t2=*new(allocate_child()) MyTask((start+end)/2, end);
  spawn(t1);
  spawn(t2);
  DoSomethingFirst();
  wait_for_all();
  DoSomethingElse();
  return 0;
}
```

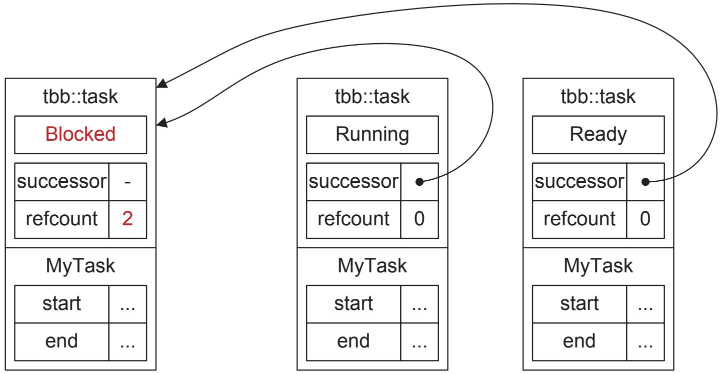


tbb::task
Running
successor -
refcount 3
MyTask
start ...
end ...

tbb::task
Created
successor
refcount 0
MyTask
start ...
end ...

tbb::task
Created
successor
refcount 0
MyTask
start ...
end ...

```
tbb::task * MyTask::execute()
{
  if(cond())
    return 0;
  set_ref_count(3);
  MyTask &t1=*new(allocate_child()) MyTask(start,(start+end)/2);
  MyTask &t2=*new(allocate_child()) MyTask((start+end)/2, end);
  spawn(t1);
  spawn(t2);
  DoSomethingFirst();
  wait_for_all();
  DoSomethingElse();
  return 0;
}
```

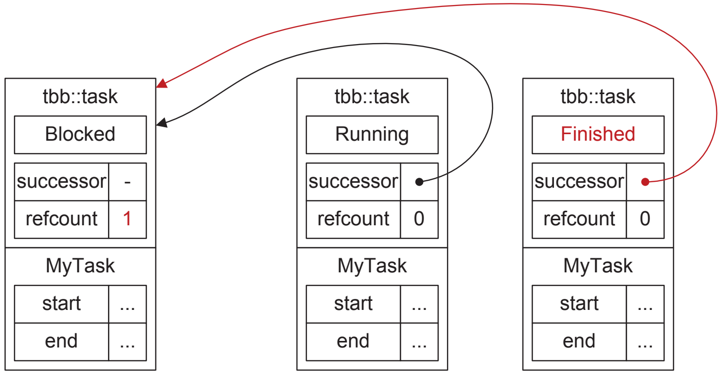


tbb::task
Blocked
successor -
refcount 2
MyTask
start ...
end ...

tbb::task
Running
successor
refcount 0
MyTask
start ...
end ...

tbb::task
Ready
successor
refcount 0
MyTask
start ...
end ...

```
tbb::task * MyTask::execute()
{
  if(cond())
    return 0;
  set_ref_count(3);
  MyTask &t1=*new(allocate_child()) MyTask(start,(start+end)/2);
  MyTask &t2=*new(allocate_child()) MyTask((start+end)/2, end);
  spawn(t1);
  spawn(t2);
  DoSomethingFirst();
  wait_for_all();
  DoSomethingElse();
  return 0;
}
```

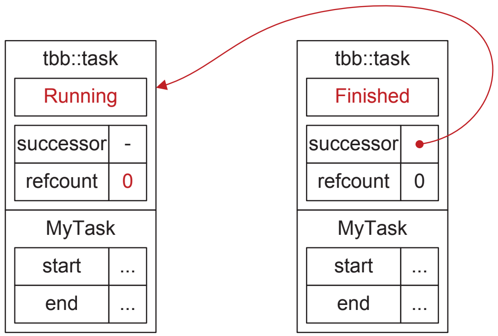


tbb::task
Blocked
successor -
refcount 1
MyTask
start ...
end ...

tbb::task
Running
successor
refcount 0
MyTask
start ...
end ...

tbb::task
Finished
successor
refcount 0
MyTask
start ...
end ...

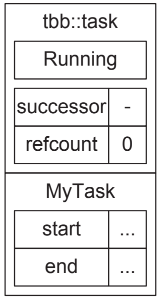
```
tbb::task * MyTask::execute()
{
  if(cond())
    return 0;
  set_ref_count(3);
  MyTask &t1=*new(allocate_child()) MyTask(start,(start+end)/2);
  MyTask &t2=*new(allocate_child()) MyTask((start+end)/2, end);
  spawn(t1);
  spawn(t2);
  DoSomethingFirst();
  wait_for_all();
  DoSomethingElse();
  return 0;
}
```



tbb::task | Running | successor | - | refcount | 0 | MyTask | start | ... | end | ...

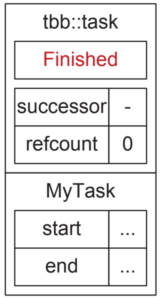
tbb::task | Finished | successor | ● | refcount | 0 | MyTask | start | ... | end | ...

```
tbb::task * MyTask::execute()
{
  if(cond())
    return 0;
  set_ref_count(3);
  MyTask &t1=*new(allocate_child()) MyTask(start,(start+end)/2);
  MyTask &t2=*new(allocate_child()) MyTask((start+end)/2, end);
  spawn(t1);
  spawn(t2);
  DoSomethingFirst();
  wait_for_all();
  DoSomethingElse();
  return 0;
}
```



tbb::task | Running | successor | - | refcount | 0 | MyTask | start | ... | end | ...

```
tbb::task * MyTask::execute()
{
  if(cond())
    return 0;
  set_ref_count(3);
  MyTask &t1=*new(allocate_child()) MyTask(start,(start+end)/2);
  MyTask &t2=*new(allocate_child()) MyTask((start+end)/2, end);
  spawn(t1);
  spawn(t2);
  DoSomethingFirst();
  wait_for_all();
  DoSomethingElse();
  return 0;
}
```

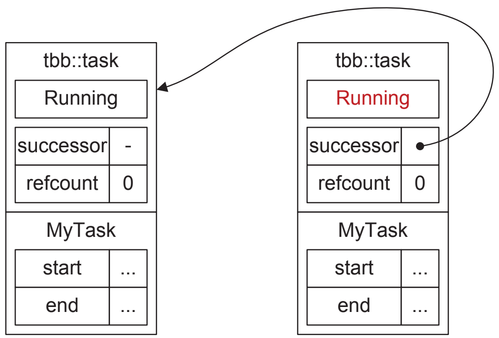


tbb::task | Finished | successor | - | refcount | 0 | MyTask | start | ... | end | ...

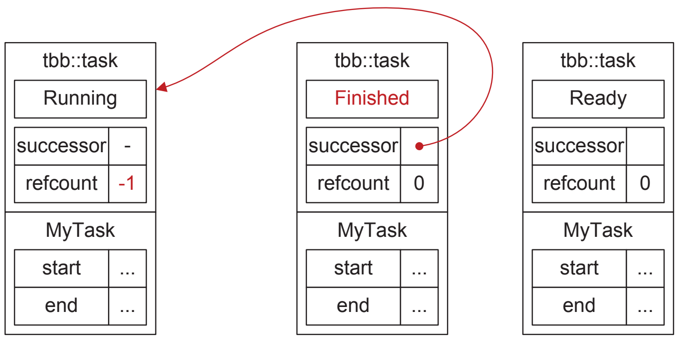
```
void CreateTasks(int start, int end)
{
   MyTask &root=*new(allocate_root()) MyTask(start,end);
   tbb::task::spawn_root_and_wait();
}
```

Managing reference counts

- What happens if we get the reference count wrong?
- Finishing task calls `decrement_ref_count` on successor
  - Automatically returns task to scheduler if count becomes zero



```
tbb::task * MyTask::execute()
{
  if(cond())
    return 0;
  MyTask &t1=*new(allocate_child()) MyTask(start,(start+end)/2);
  MyTask &t2=*new(allocate_child()) MyTask((start+end)/2, end);
  spawn(t1);
  spawn(t2);
  set_ref_count(3);
  DoSomethingFirst();
  wait_for_all();
  DoSomethingElse();
  return 0;
}
```



```
tbb::task * MyTask::execute()
{
  if(cond())
    return 0;
  MyTask &t1=*new(allocate_child()) MyTask(start,(start+end)/2);
  MyTask &t2=*new(allocate_child()) MyTask((start+end)/2, end);
  spawn(t1);
  spawn(t2);
  set_ref_count(3);
  DoSomethingFirst();
  wait_for_all();
  DoSomethingElse();
  return 0;
}
```

Some help is available

- TBB library comes in two forms: debug and release
  - release library does no error checking – all about speed
  - debug library will check reference counts at many points
- Choose library version at compilation and link stages
  - Debug: #define TBB_USE_DEBUG=1 when compiling
    - On microsoft compilers it will automatically link the correct library
    - On other compilers use "`-ltbb`" vs "`-ltbb_debug`"
  - Usually maintain different release and debug settings
    - Debug: /DTBB_USE_DEBUG=1 /MDd
    - Release: /DNDEBUG=1 /O2
  - Can setup in Visual Studio or in a makefile

Data-parallelism vs task parallelism

- Two very broad types of parallelism we've seen so far
- **Data-parallelism**: *do the same task lots of time in any order*
  - The code for the task stays the same for each execution
  - The input to the task changes with each execution
  - There are no dependencies between different executions
    - Often applied to elements of an array
    - Also described as "loop-parallelism"
- **Task-parallelism**: *do many different tasks with dependencies*
  - Each task has zero or more dependencies that must be met
  - Different tasks ***may*** have different code
  - More powerful than data-parallelism?