Recap: Overview of task groups

- A task group collects together a number of child tasks
  - The task creating the group is called the parent
  - One or more child tasks are created and `run()` by the parent
  - Child tasks ***may*** execute in parallel
  - Parent task must `wait()` for all child tasks before returning

- Some important differences between tasks and threads
  - Threads ***must*** execute in parallel
  - A thread may continue after its creator exits
  - Threads must be joined individually

parallel_for using `tbb::task_group`

```
#include "tbb/task_group.h"

template<class TI, class TF>
void parallel_for(const TI &begin, const TI &end, const TF &f)
{
   if(begin+1 == end){
     f(begin);
   }else{
     auto left=[&](){  parallel_for(begin, (begin+end)/2,  f); }
     auto right=[&](){ parallel_for((begin+end)/2, end,     f); }

     // Spawn the two tasks in a group
     tbb::task_group group;
     group.run(left);
     group.run(right);

     group.wait(); // Wait for both to finish
   }
}
```

More patterns: `tbb::parallel_invoke`

```
template<typename Func0, typename Func1>
void parallel_invoke(const Func0& f0, const Func1& f1);

template<typename Func0, typename Func1, typename Func2>
void parallel_invoke(const Func0& f0, const Func1& f1, const Func2& f2);
```

- Takes two or more functions and ***may*** run in parallel
  - Overloaded for different numbers of arguments
  - No overload for 1 argument for obvious reasons
- Interface is very clean, but also quite simple
  - Decision about number of tasks is completely static
  - You can’t add more tasks once some starts
  - No choice about when to synchronise with tasks

parallel_invoke using task_group

- parallel_invoke can be implement using task_group
  - task_group supports a super-set of the functionality

```
template<typename Fc0, typename Fc1, typename Fc2>
void parallel_invoke(const Fc0& f0, const Fc1& f1, const Fc2& f2)
{
   tbb::task_group group;
   group.run(f0);
   group.run(f1);
   group.run(f2);
   group.wait();
}
```

Can't[1] do task_group using parallel_invoke

- task_group is intrinsically dynamic
  - Decide how much work to add at run-time
  - Can add work even while tasks are running in the group

```
void my_function(int n, float *x)
{
    tbb::task_group group;
    for(unsigned i=0;i<n;i++){
        if(x[i]==0)
            group.run([=](){  f(i); });
        else
            group.run([=](){  g(x[i]); });
    }
    group.wait();
}
```

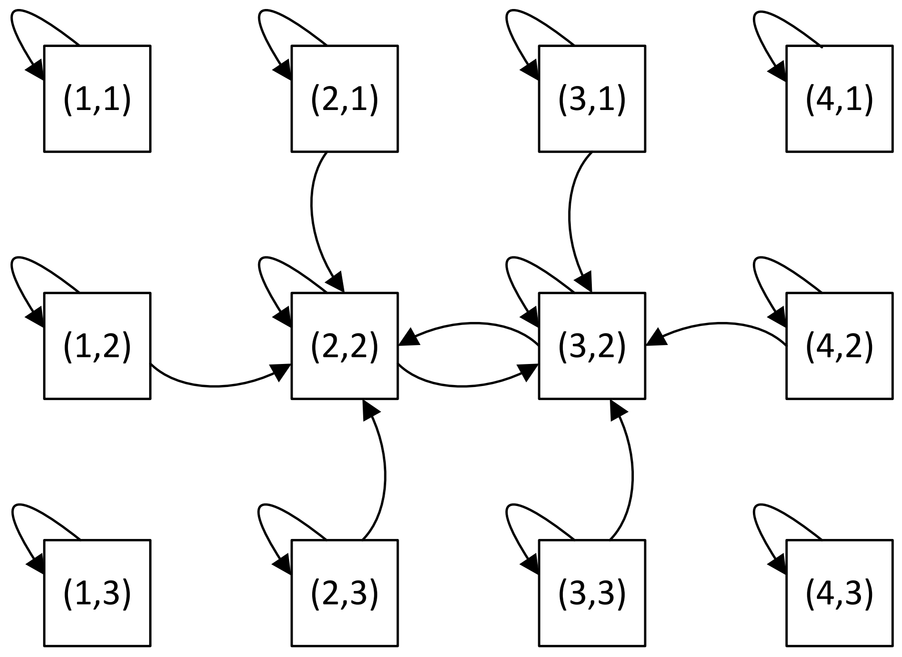
[1] - Easily

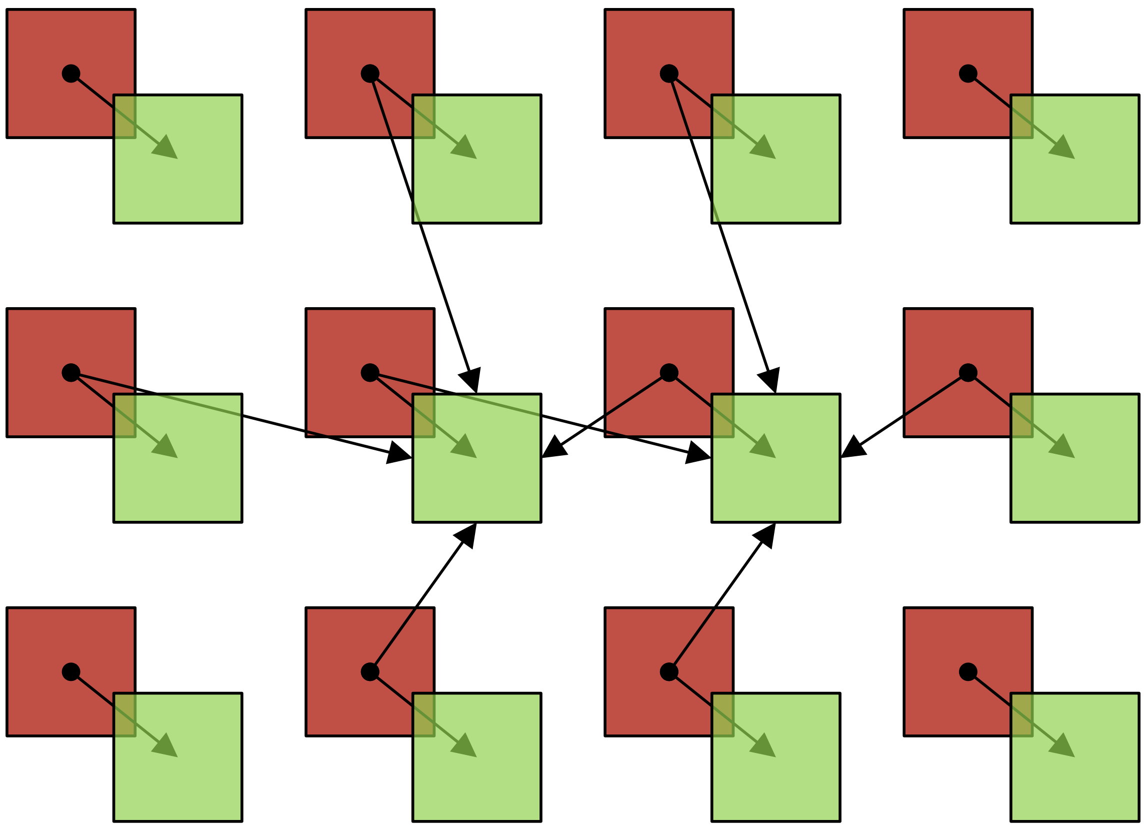
More complex loop nests

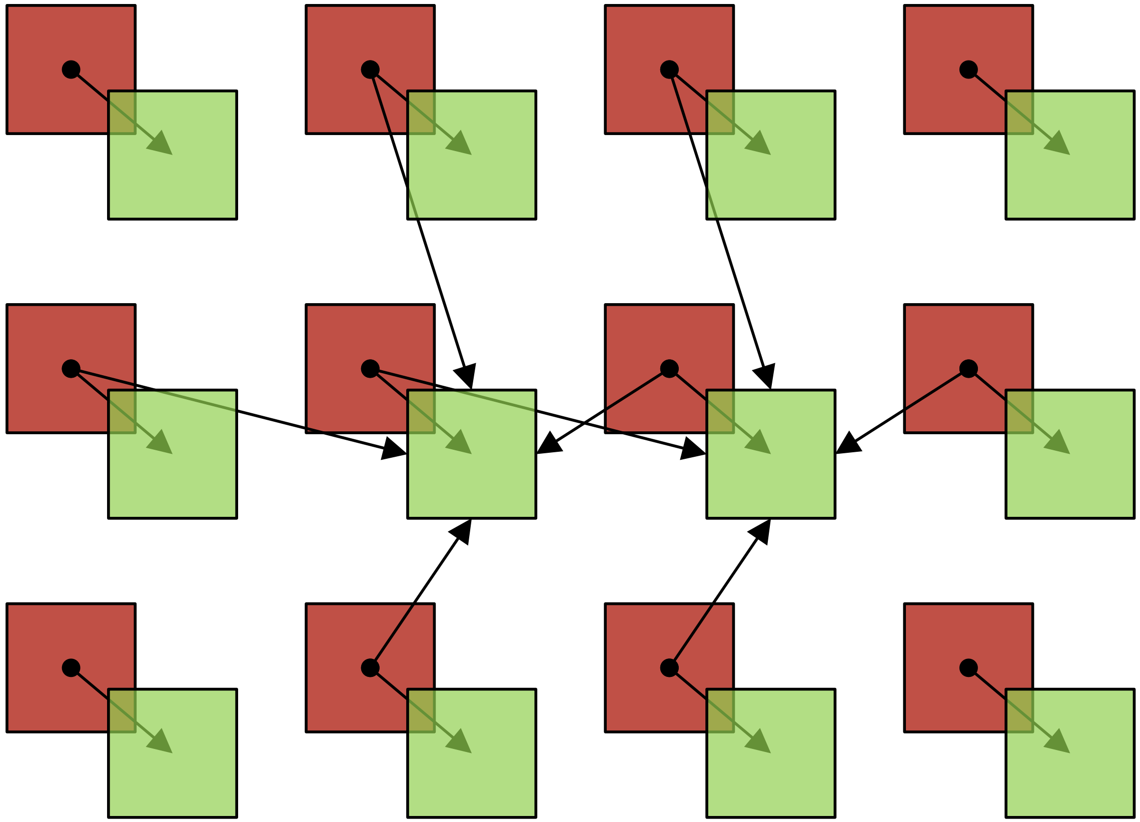
```
std::vector<uint8_t> process_frame(
    int n, int w, int h,
    std::vector<uint8_t> fIn // Receive frame (by value)
){
    // Handle n==0 case
    std::vector<uint8_t> fOut=fIn;

    for(int i=1; i<n; i++){
        fIn = fOut;

        for(int y=1; y < h-1; y++){
            for(int x=1; x < w-1; x++){
                uint8_t nhood [5] = {
                               fIn[(y-1)*w+x],
                fIn[y*w+(x-1)],fIn[ y   *w+x], fIn[y*w+(x+1)],
                               fIn[(y+1)*w+x]
                };
                uint8_t value = min_of_array(5, nhood);
                fOut[y*w+x] = value;
            }
        }
    }
    return fOut;
}
```

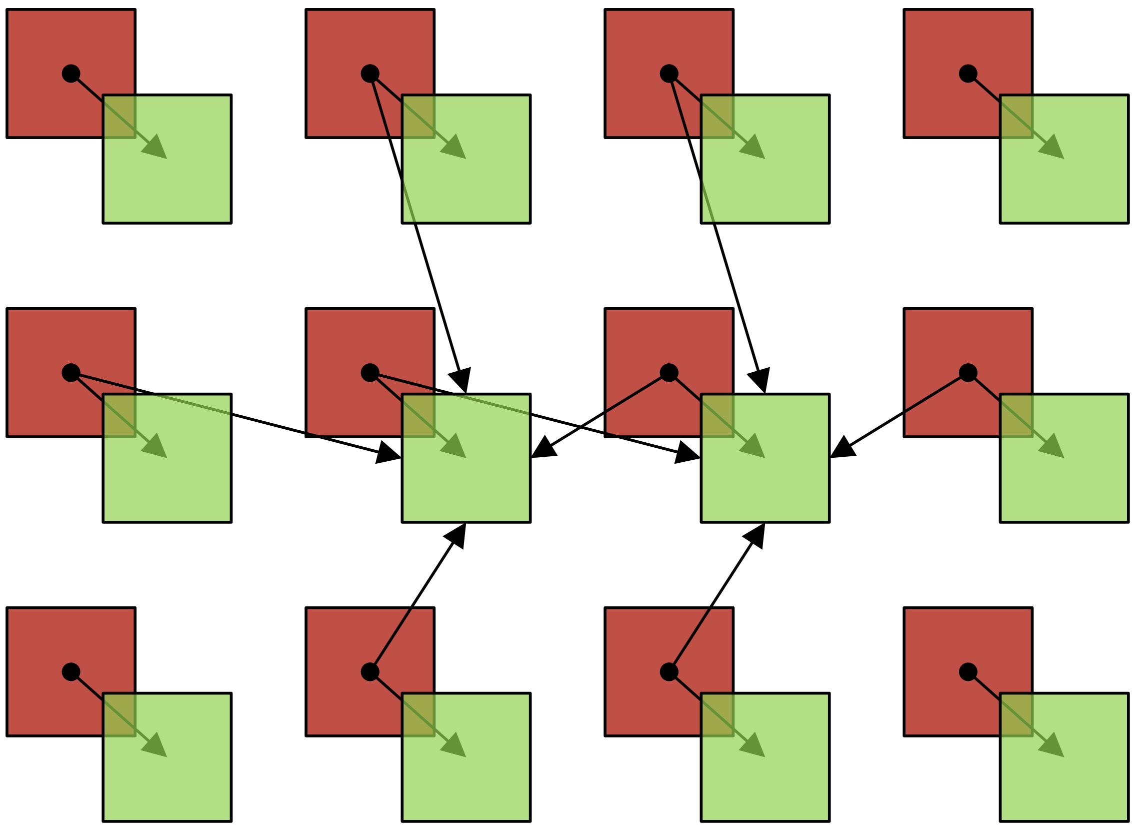
How do you parallelise?

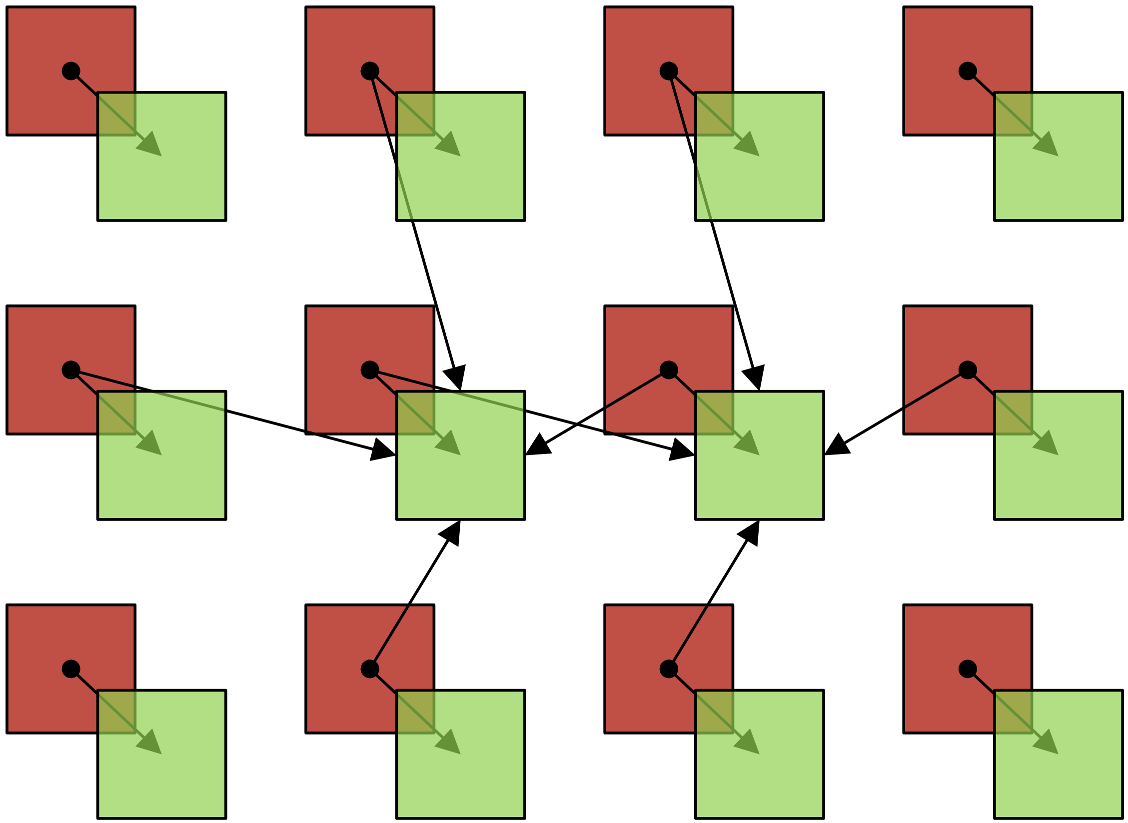
```
uint8_t nhood[5] = {
                  fIn[(y-1)*w+x],
  fIn[y*w+(x-1)],fIn[(y+0)*w+x],fIn[y*w+(x+1)],
                  fIn[(y+1)*w+x]
};

uint8_t value = min_of_array(5, nhood);

fOut[y*w+x] = value;
```

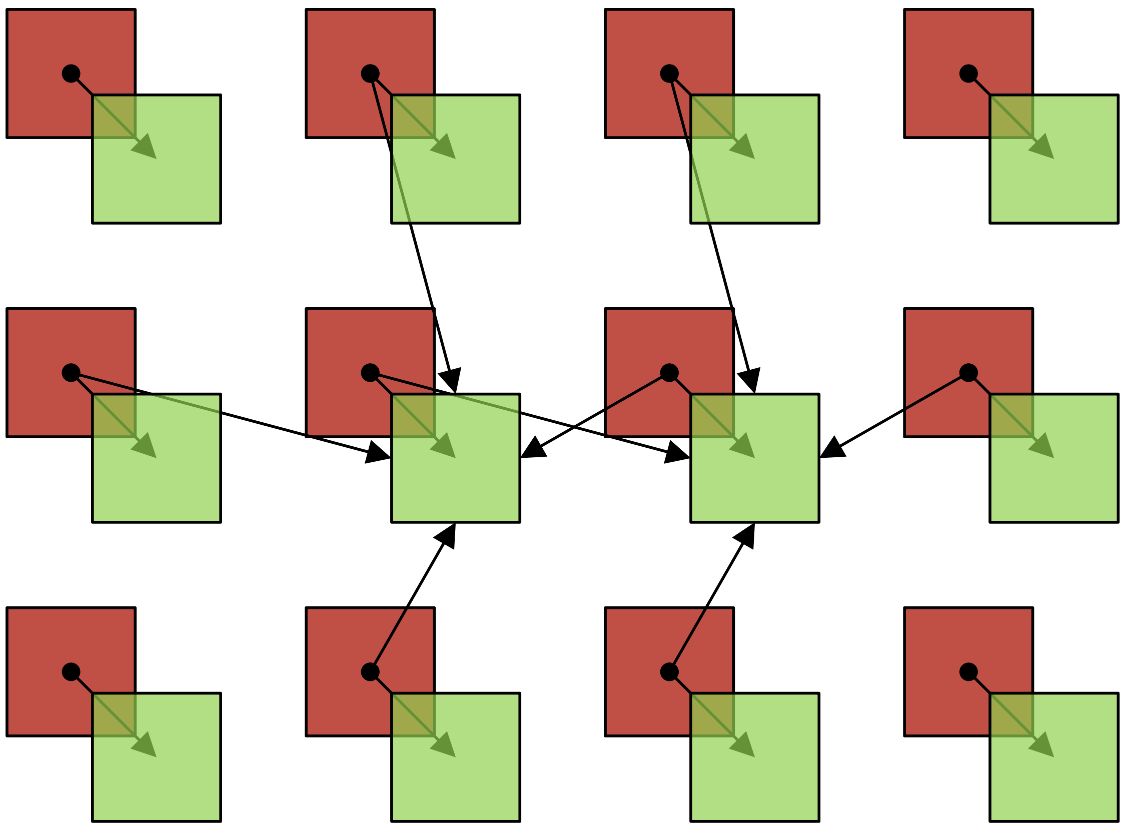


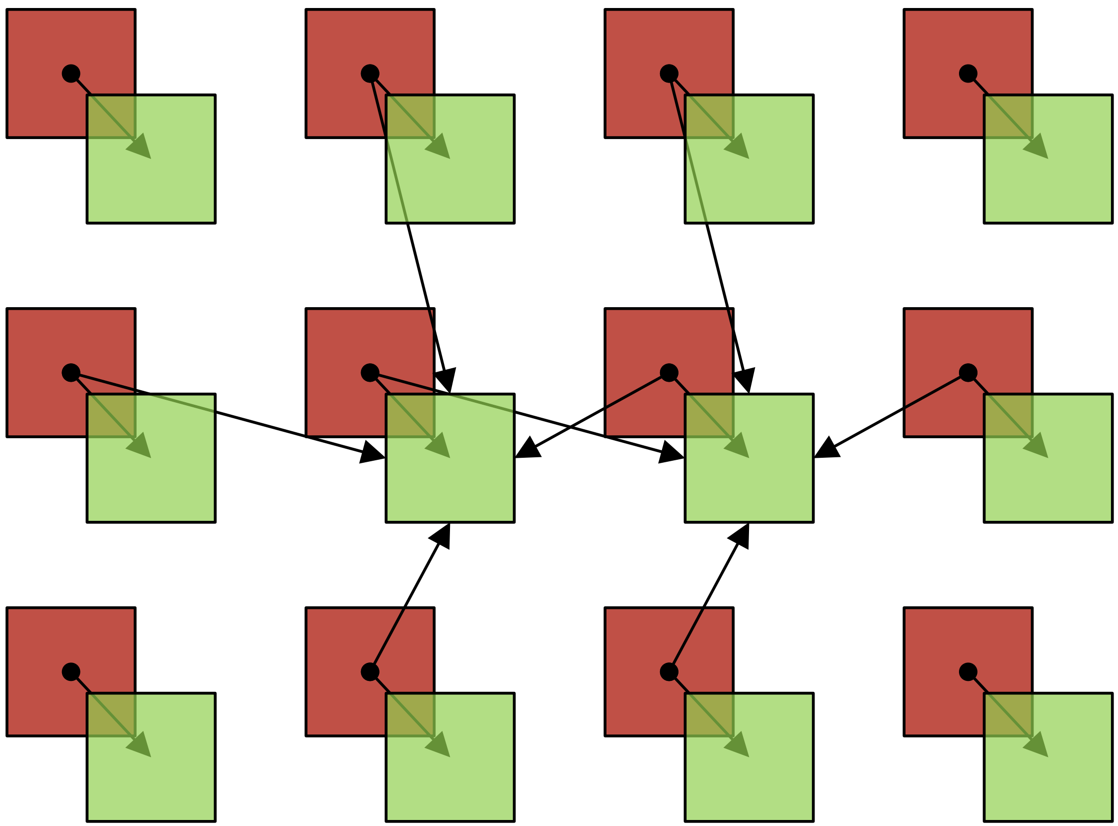












```
for(int i=1; i<n; i++){
    fIn = fOut;

    for(int y=1; y < h-1; y++){
        for(int x=1; x < w-1; x++){
            uint8_t nhood [5] = {
                               fIn[(y-1)*w+x],
            fIn[y*w+(x-1)],fIn[ y   *w+x], fIn[y*w+(x+1)],
                               fIn[(y+1)*w+x]
            };
            uint8_t value = min_of_array(5, nhood);
            fOut[y*w+x] = value;
        }
    }
}
```

- Can vectorise over both x and y
- Intermediate variable tmp saves us

Parallelising over a 2d space

```
for(int i=0; i<n; i++){
    fIn = fOut;

    tbb::blocked_range2d<int> r( 1,h-1, 1,w-1 );

    tbb::parallel_for( r, [&](const tbb::blocked_range2d<int> &xy) {
        for(int y=xy.rows().begin(); y < xy.rows().end(); y++){
            for(int x=xy.cols().begin(); x < xy.cols().end(); x++){
                uint8_t nhood [5] = {
                               fIn[(y-1)*w+x],
                fIn[y*w+(x-1)],fIn[ y   *w+x], fIn[y*w+(x+1)],
                               fIn[(y+1)*w+x]
                };

                uint8_t value = min_of_array(5, nhood);

                fOut[y*w+x] = value;
            }
        }
    });
}
```

Outer loop?

```
for(int i=0; i<n; i++){
    fIn = fOut;

    tbb::blocked_range2d<int> r( 1,h-1, 1,w-1 );

    tbb::parallel_for( r, [&](const tbb::blocked_range2d<int> &xy) {
        for(int y=xy.rows().begin(); y < xy.rows().end(); y++){
            for(int x=xy.cols().begin(); x < xy.cols().end(); x++){
                uint8_t nhood [5] = {
                                  fIn[(y-1)*w+x],
                fIn[y*w+(x-1)],fIn[ y   *w+x], fIn[y*w+(x+1)],
                                  fIn[(y+1)*w+x]
                };

                uint8_t value = min_of_array(5, nhood);

                fOut[y*w+x] = value;
            }
        }
    });
}
```

- Strict loop carried dependency
- Pure cyclic chain
  - Impossible to break
- No parallelism...?
- We'll come back to it

The underlying primitive: tbb::task

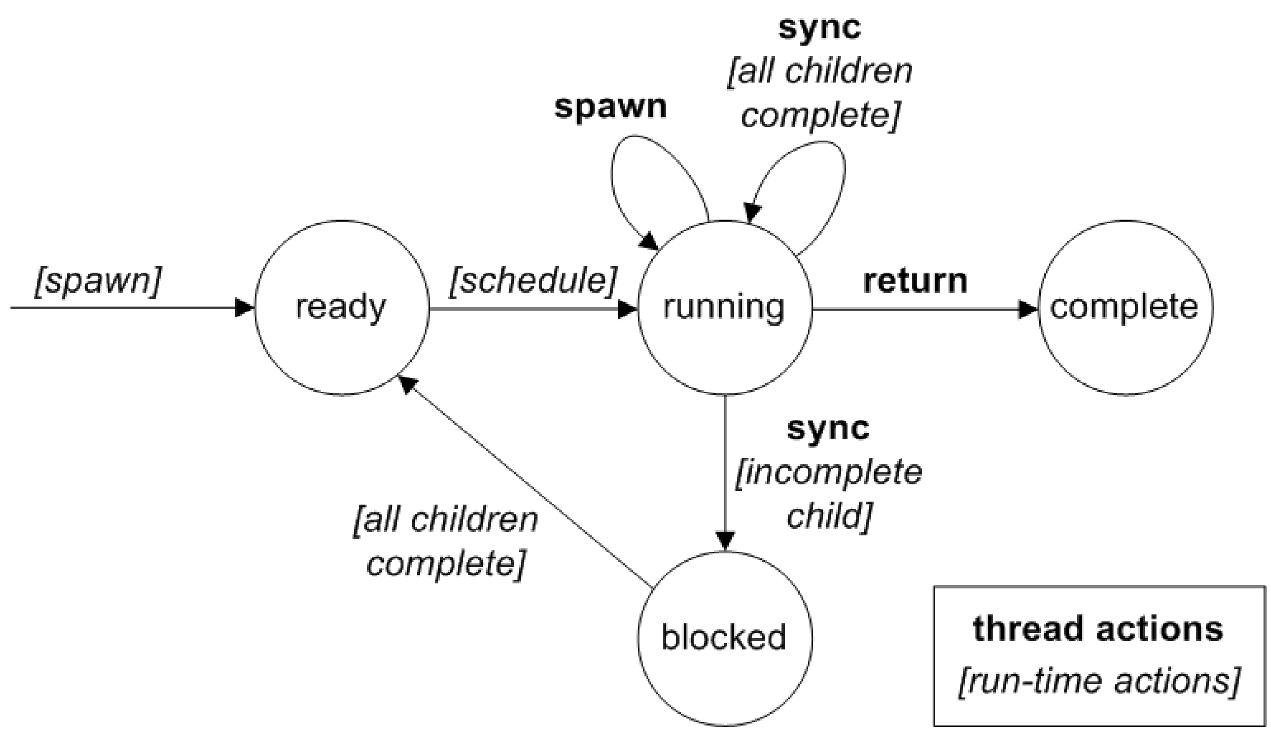
- TBB has a basic primitive called tbb::task
- This is the raw unit of scheduling understood by the lib.
  - Other high-level wrappers create tasks internally
  - The TBB run-time takes tasks and schedules them to a CPU
- Tasks are very flexible, with a lot of power
  - Can express complicated dependency graphs
  - Build non-local synchronisation and barriers
- With power comes responsibility
  - They allow you to make mistakes
  - Possible (though not likely) to mess up the TBB run-time
- Better to create wrappers on top that hide tasks
  - parallel_for, parallel_invoke, task_group, parallel_reduce, ...

```
void MyTask(int start, int end)
{
 if(cond())
   return 0;
 tbb::task_group group;
 group.run([=](){MyTask(start,(start+end)/2); });
 group.run([=](){MyTask((start+end)/2,end); });
 DoSomethingFirst();
 group.wait();
 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();
}
```

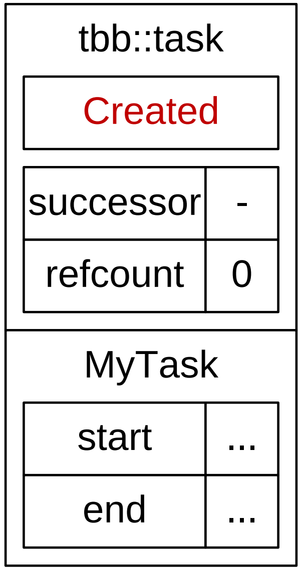
Life-cycle of a task

- Life-cycle of task due to interaction between task and run-time
  - Individual task calls **spawn**, **wait_for_all (sync)**, **return**
  - TBB run-time will keep track of a task's children (*dependencies*)

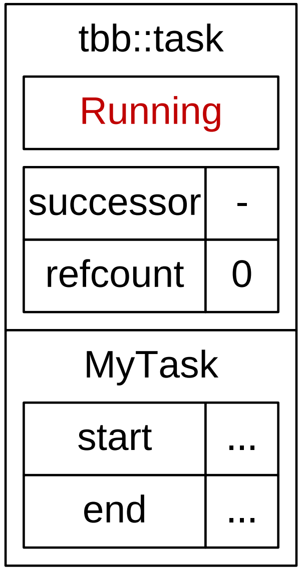


Scheduling through reference counts

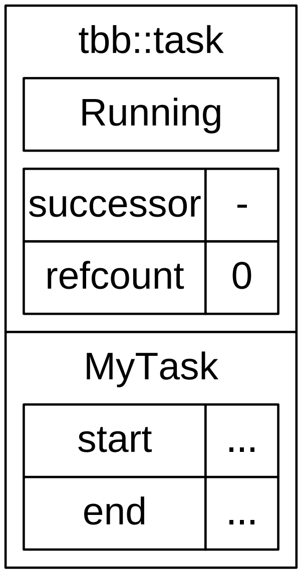
- Each task has a reference count and a successor task
- The reference count identifies whether a task is blocked
  - If the reference count is zero then the task could be run
  - But only if it has been given to the task scheduler
  - Legal to create a task and not give it to the scheduler
  - *Note the difference: "reference count" vs "C++ reference"*
- Successor task identifies the task blocked by this task
  - Generally the successor task is the creator, or parent
  - When a task completes it decrements the count of its successor



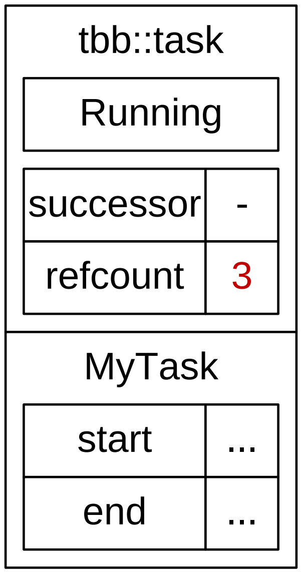

```
void CreateTasks(int start, int end)
{
   MyTask &root=*new(allocate_root()) MyTask(start,end);
   tbb::task::spawn_root_and_wait();
}
```



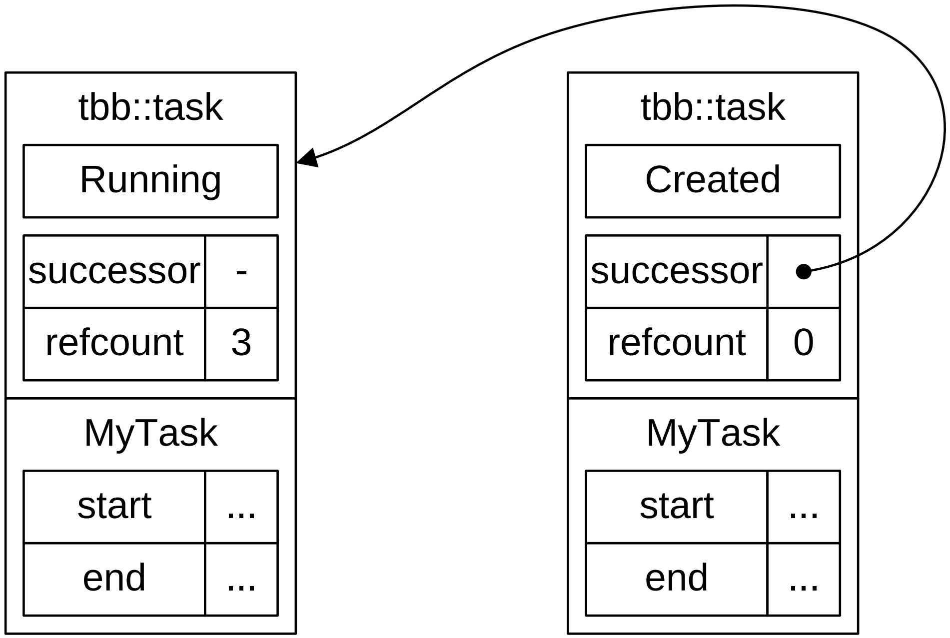

```
void CreateTasks(int start, int end)
{
   MyTask &root=*new(allocate_root()) MyTask(start,end);
   tbb::task::spawn_root_and_wait();
}
```



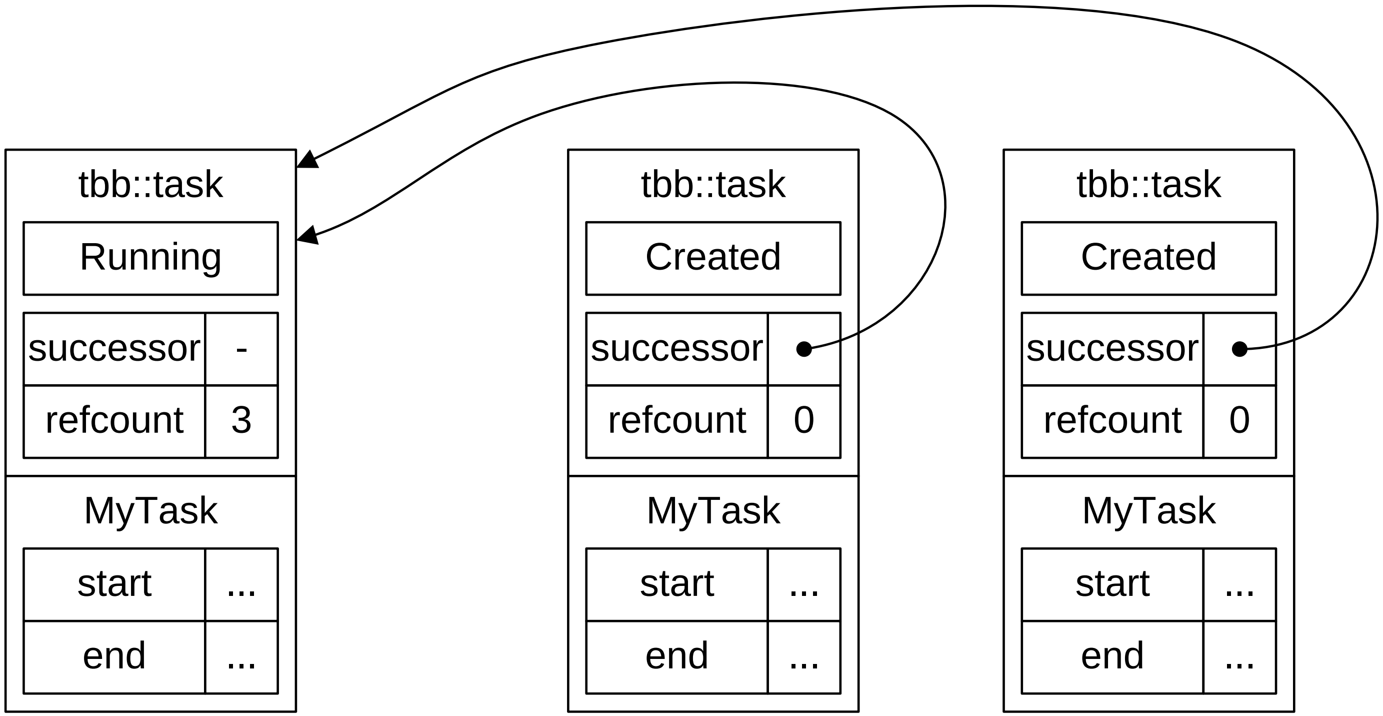

```
tbb::task * MyTask::execute()
{
  if(cond())
    return 0;
  set_ref_count(3);
  MyTask &t1=*new(allocate_child()) MyTask(start,(start+end)/2);
  MyTask &t2=*new(allocate_child()) MyTask((start+end)/2, end);
  spawn(t1);
  spawn(t2);
  DoSomethingFirst();
  wait_for_all();
  DoSomethingElse();
 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



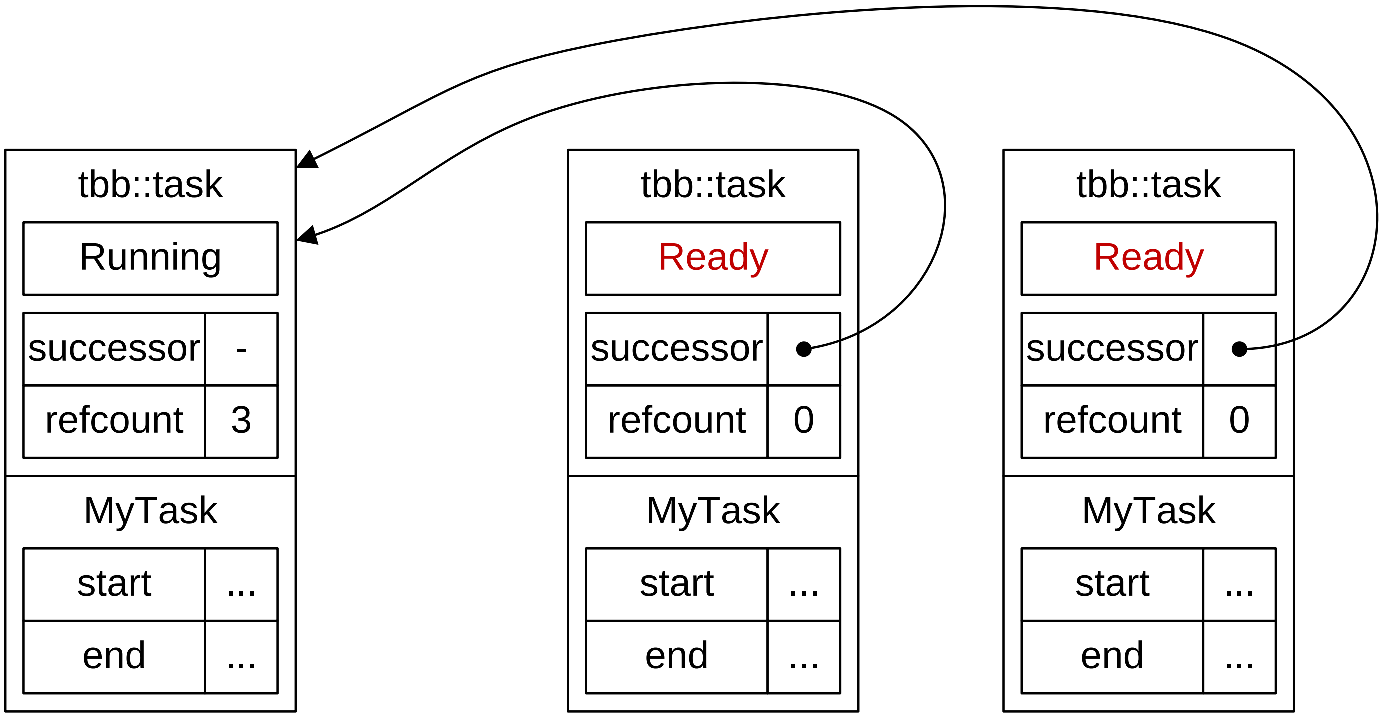

```
tbb::task * MyTask::execute()
{
  if(cond())
    return 0;
  set_ref_count(3);
  MyTask &t1=*new(allocate_child()) MyTask(start,(start+end)/2);
  MyTask &t2=*new(allocate_child()) MyTask((start+end)/2, end);
  spawn(t1);
  spawn(t2);
  DoSomethingFirst();
  wait_for_all();
  DoSomethingElse();
 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



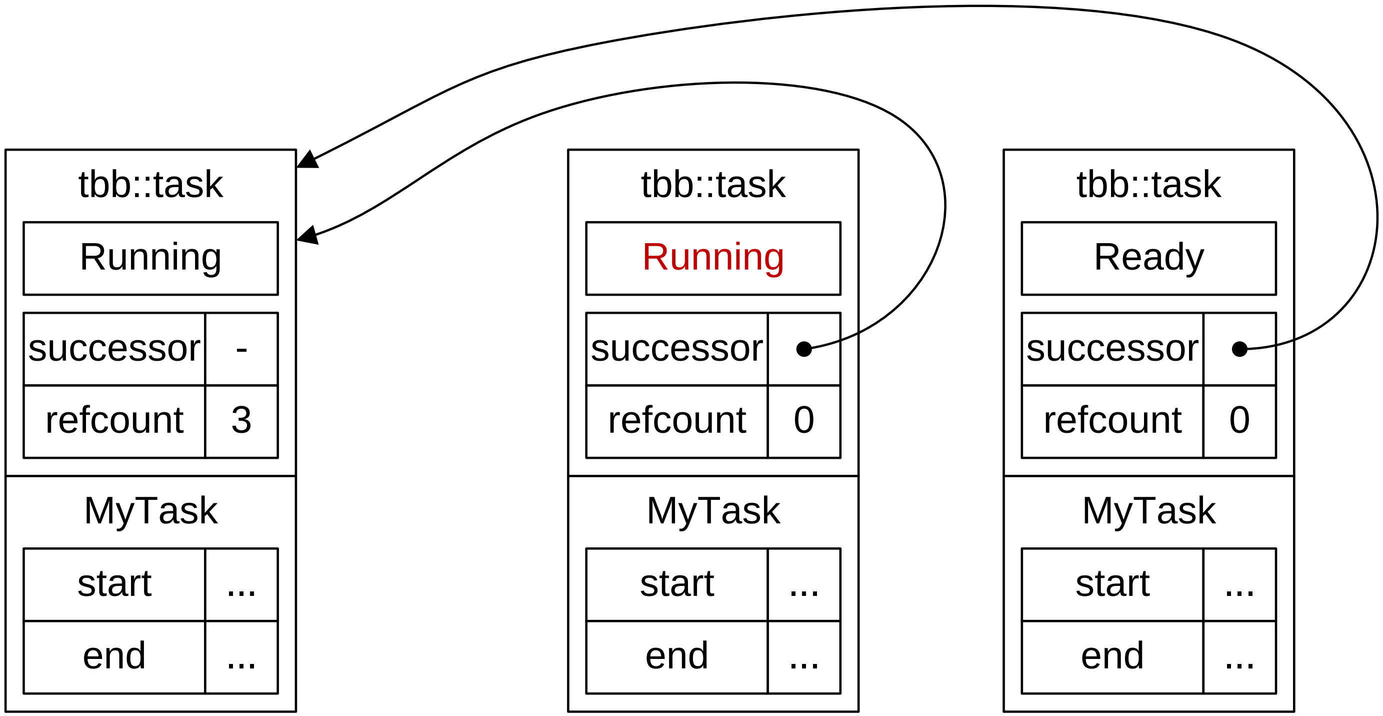

```
tbb::task * MyTask::execute()
{
  if(cond())
    return 0;
  set_ref_count(3);
  MyTask &t1=*new(allocate_child()) MyTask(start,(start+end)/2);
  MyTask &t2=*new(allocate_child()) MyTask((start+end)/2, end);
  spawn(t1);
  spawn(t2);
  DoSomethingFirst();
  wait_for_all();
  DoSomethingElse();
 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



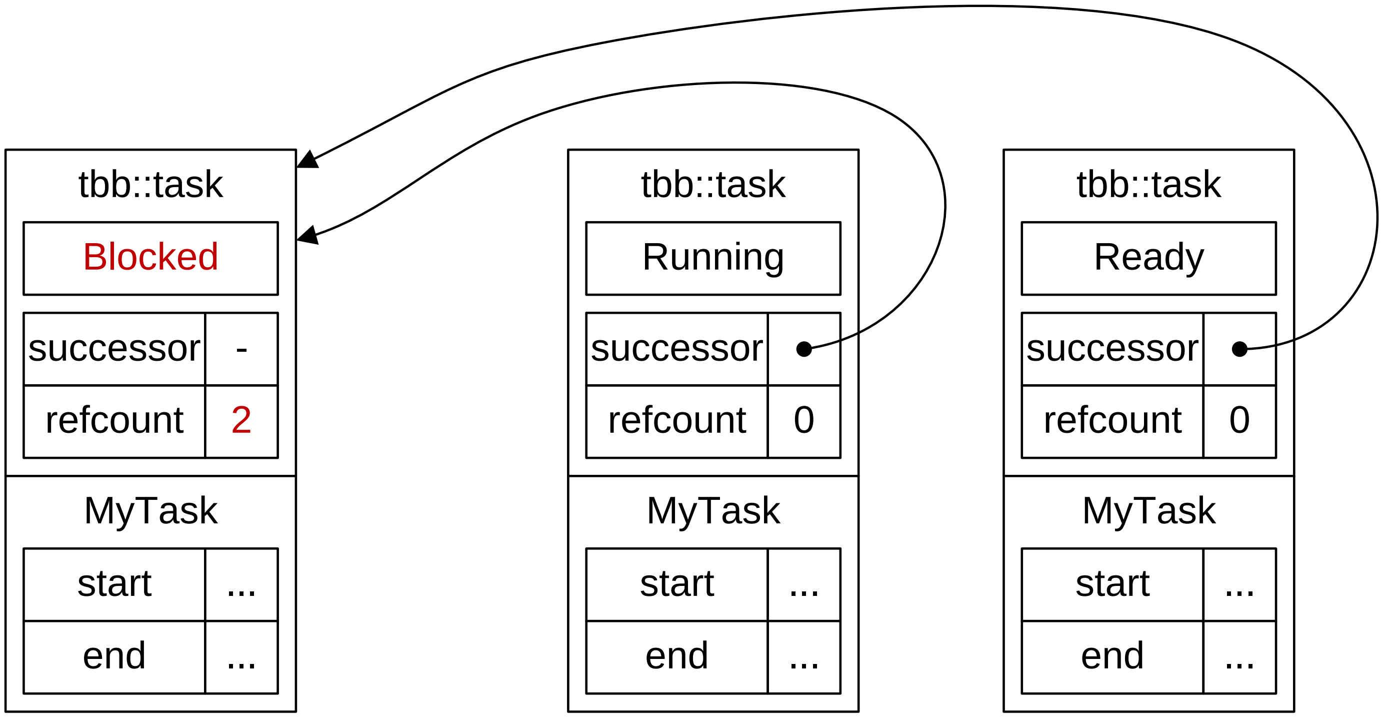

```
tbb::task * MyTask::execute()
{
  if(cond())
    return 0;
  set_ref_count(3);
  MyTask &t1=*new(allocate_child()) MyTask(start,(start+end)/2);
  MyTask &t2=*new(allocate_child()) MyTask((start+end)/2, end);
  spawn(t1);
  spawn(t2);
  DoSomethingFirst();
  wait_for_all();
  DoSomethingElse();
 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



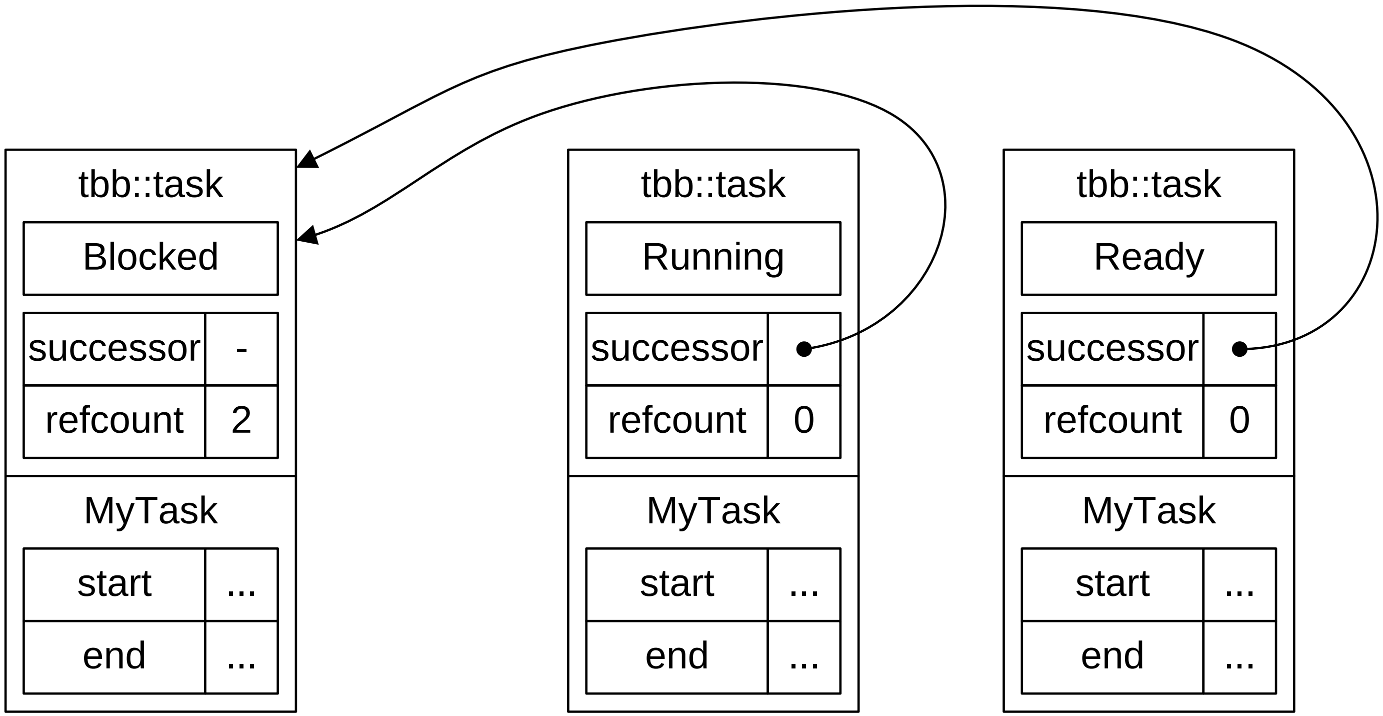

```
tbb::task * MyTask::execute()
{
  if(cond())
    return 0;
  set_ref_count(3);
  MyTask &t1=*new(allocate_child()) MyTask(start,(start+end)/2);
  MyTask &t2=*new(allocate_child()) MyTask((start+end)/2, end);
  spawn(t1);
  spawn(t2);
  DoSomethingFirst();
  wait_for_all();
  DoSomethingElse();
 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



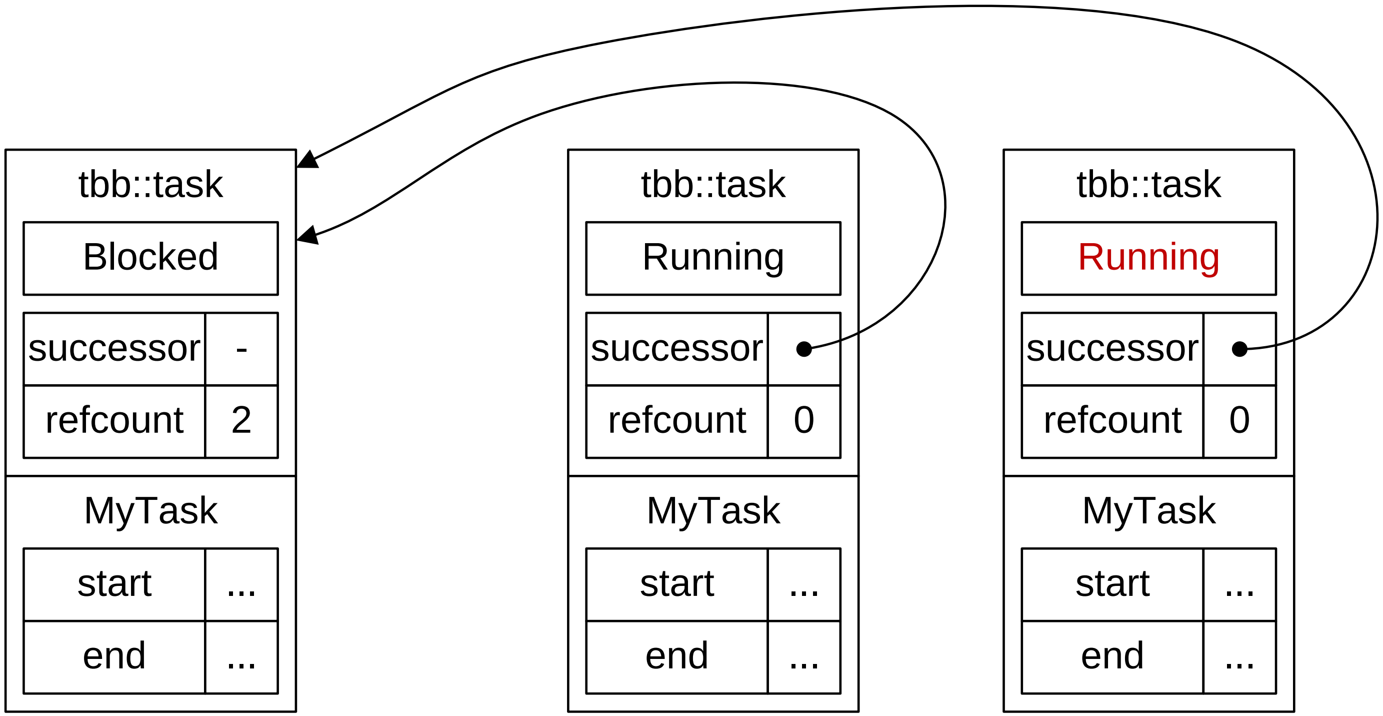

```
tbb::task * MyTask::execute()
{
  if(cond())
    return 0;
  set_ref_count(3);
  MyTask &t1=*new(allocate_child()) MyTask(start,(start+end)/2);
  MyTask &t2=*new(allocate_child()) MyTask((start+end)/2, end);
  spawn(t1);
  spawn(t2);
  DoSomethingFirst();
  wait_for_all();
  DoSomethingElse();
 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



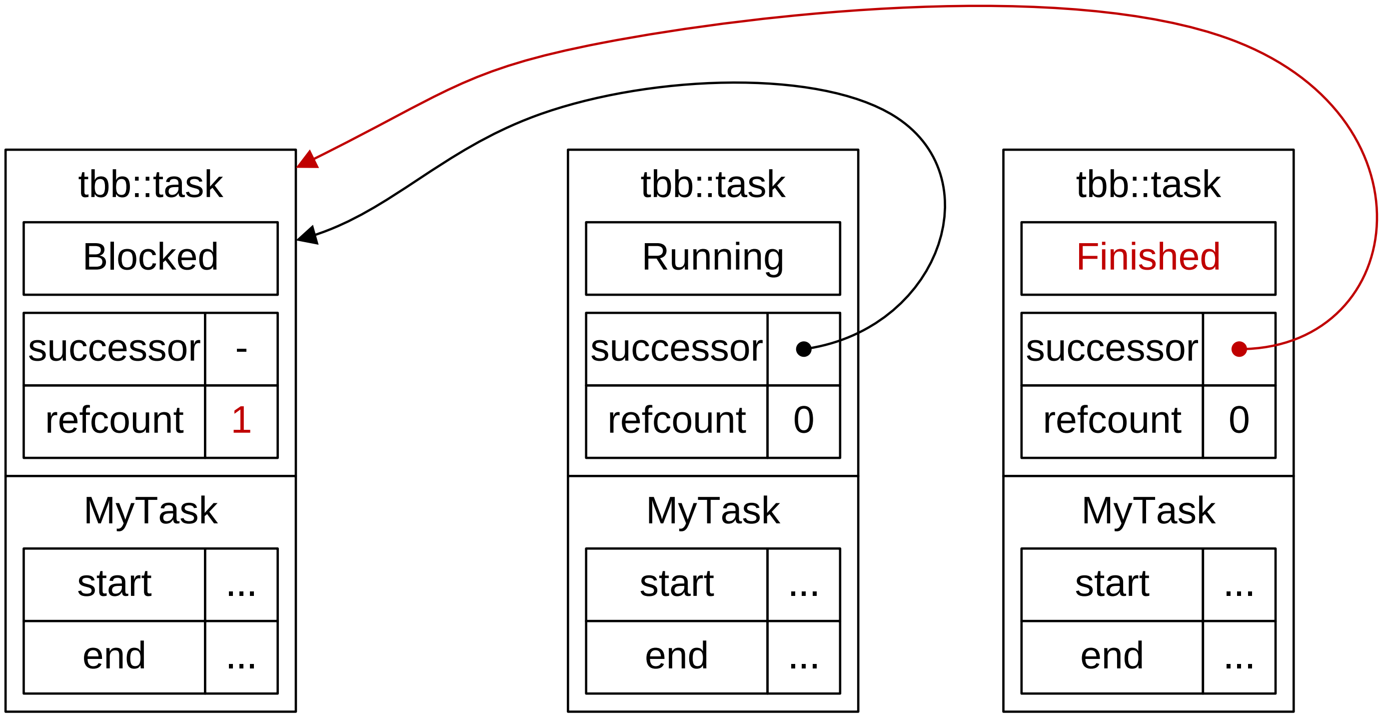

```
tbb::task * MyTask::execute()
{
  if(cond())
    return 0;
  set_ref_count(3);
  MyTask &t1=*new(allocate_child()) MyTask(start,(start+end)/2);
  MyTask &t2=*new(allocate_child()) MyTask((start+end)/2, end);
  spawn(t1);
  spawn(t2);
  DoSomethingFirst();
  wait_for_all();
  DoSomethingElse();
 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



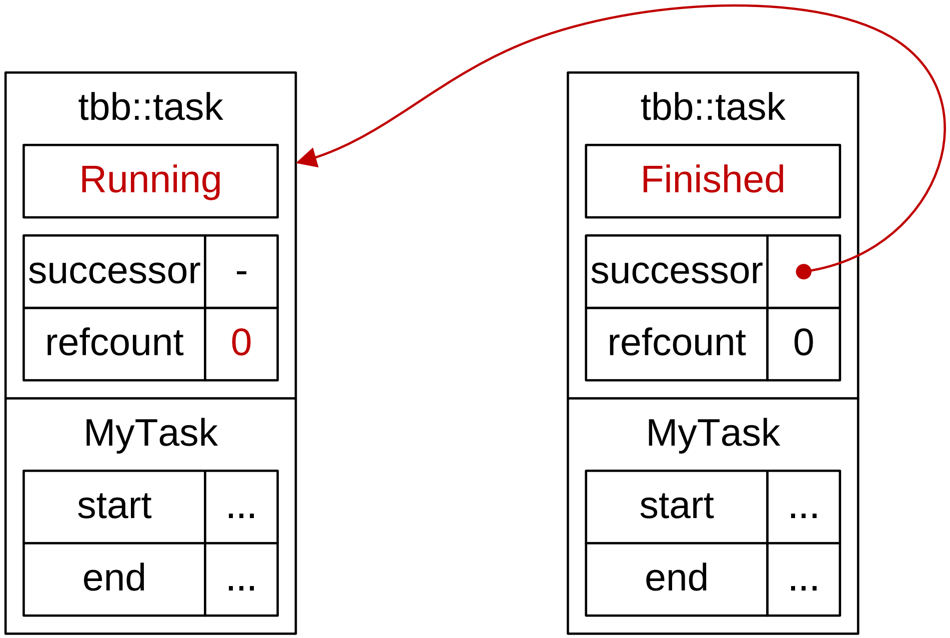

```
tbb::task * MyTask::execute()
{
  if(cond())
    return 0;
  set_ref_count(3);
  MyTask &t1=*new(allocate_child()) MyTask(start,(start+end)/2);
  MyTask &t2=*new(allocate_child()) MyTask((start+end)/2, end);
  spawn(t1);
  spawn(t2);
  DoSomethingFirst();
  wait_for_all();
  DoSomethingElse();
 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



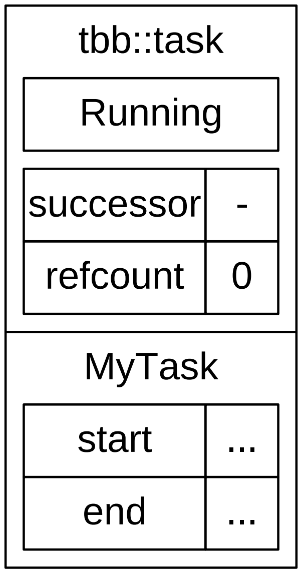

```
tbb::task * MyTask::execute()
{
  if(cond())
    return 0;
  set_ref_count(3);
  MyTask &t1=*new(allocate_child()) MyTask(start,(start+end)/2);
  MyTask &t2=*new(allocate_child()) MyTask((start+end)/2, end);
  spawn(t1);
  spawn(t2);
  DoSomethingFirst();
  wait_for_all();
  DoSomethingElse();
 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



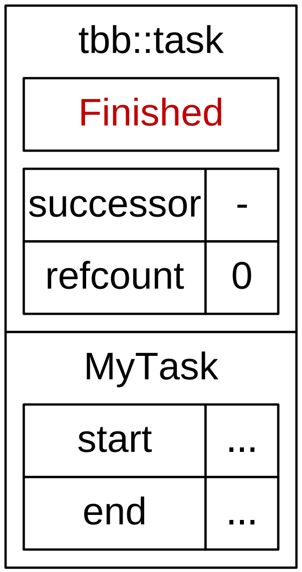

```
tbb::task * MyTask::execute()
{
  if(cond())
    return 0;
  set_ref_count(3);
  MyTask &t1=*new(allocate_child()) MyTask(start,(start+end)/2);
  MyTask &t2=*new(allocate_child()) MyTask((start+end)/2, end);
  spawn(t1);
  spawn(t2);
  DoSomethingFirst();
  wait_for_all();
  DoSomethingElse();
 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




```
tbb::task * 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 * 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;
}
```

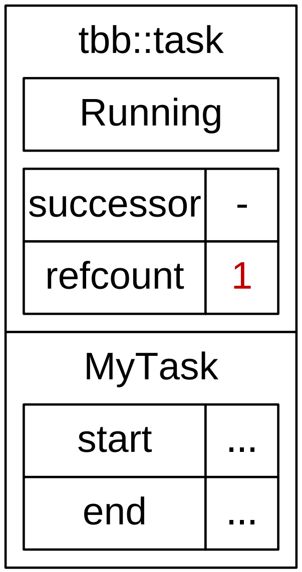



```
void CreateTasks(int start, int end)
{
   MyTask &root=*new(allocate_root()) MyTask(start,end);
   tbb::task::spawn_root_and_wait();
}
```

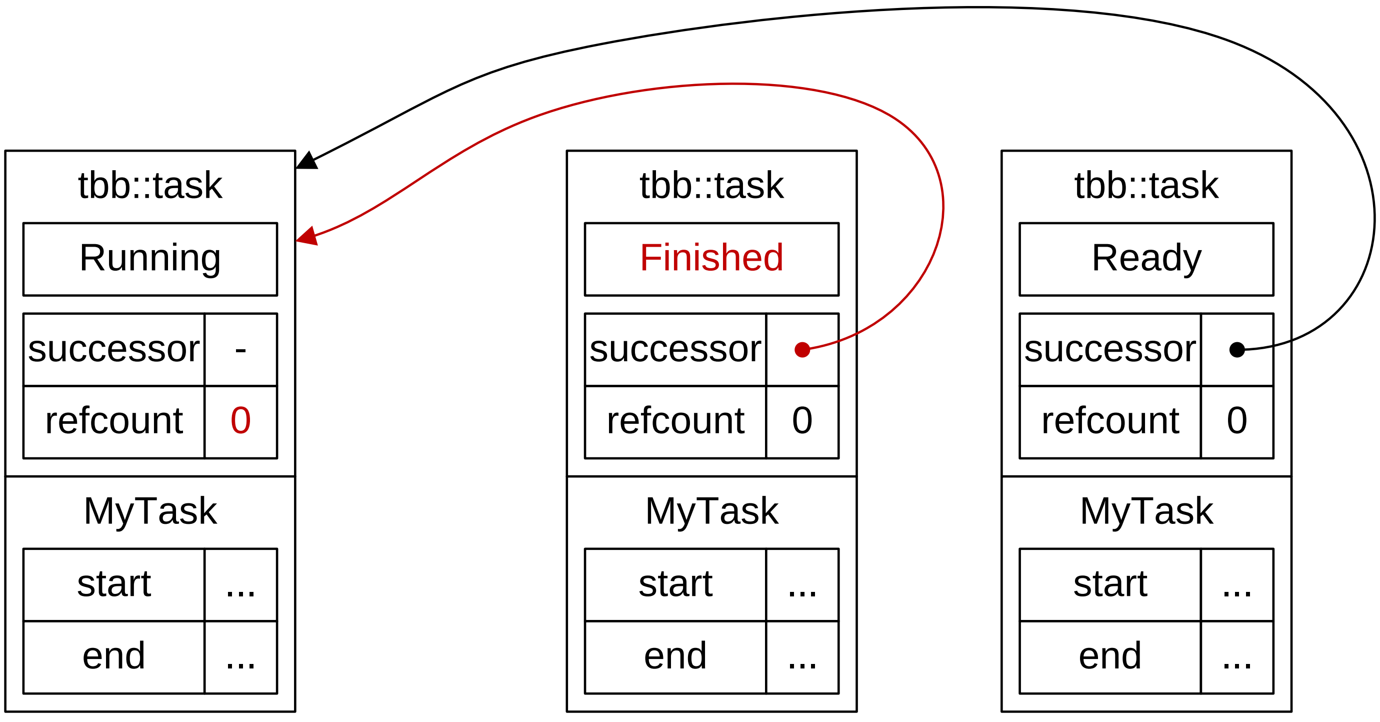
Managing reference counts

- What happens if we get the reference count wrong?

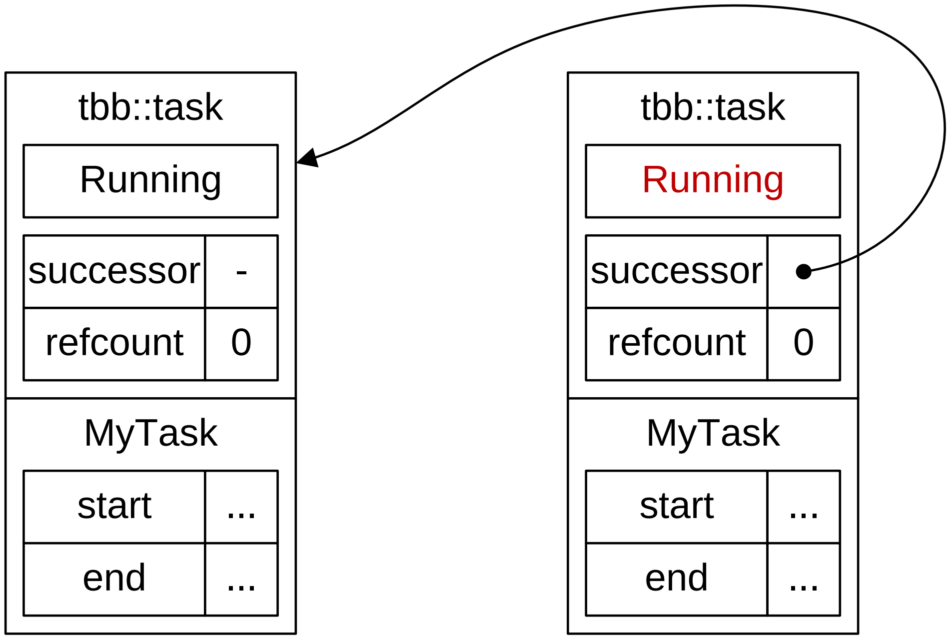
- Finishing task calls `decrement_ref_count` on successor
  - Automatically returns task to scheduler if count becomes zero



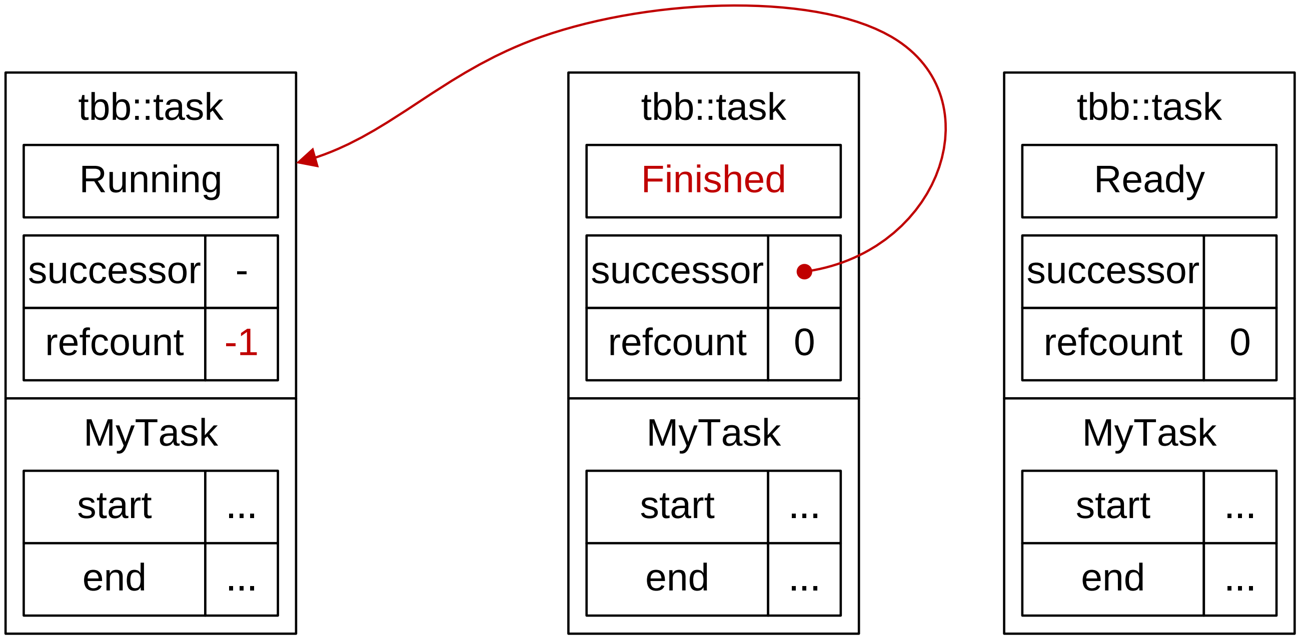

```
tbb::task * MyTask::execute()
{
  if(cond())
    return 0;
  set_ref_count(1);
  MyTask &t1=*new(allocate_child()) MyTask(start,(start+end)/2);
  MyTask &t2=*new(allocate_child()) MyTask((start+end)/2, end);
  spawn(t1);
  spawn(t2);
  DoSomethingFirst();
  wait_for_all();
  DoSomethingElse();
  return 0;
}
```




```
tbb::task * MyTask::execute()
{
  if(cond())
    return 0;
  set_ref_count(1);
  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 * 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;
}
```

Many design patterns are built on tasks

- Iteration in various forms
  - parallel_for, parallel_for_each
- Reduction and accumulation
  - parallel_reduce
- Data-dependent looping and queue processing
  - parallel_do
- Support for heterogeneous tasks
  - parallel_invoke, task_group
- Heterogeneous tasks and token-based data-flow
  - parallel_pipeline
- Goal: turn design patterns in concrete functions