Goals of TBB

- In-process shared-memory parallelism
  - Can directly pass pointers to data-structures between tasks
- Sophisticated design-patterns
  - Data-parallelism and pipeline-parallelism and task parallelism ...
- Robust ***and*** efficient constructs to support design-patterns
  - Try to make it difficult to write programs that don't work
- Efficient scheduling of tasks to CPUs
  - Try to automatically match active tasks to number of cores
- Good interoperability with the existing world
  - Can be used within existing applications and code bases

Threaded Building Blocks : parallel_for

```
void erode(
  unsigned w, unsigned h,
  const uint8_t *pSrc,
  uint8_t *pDst
){
  for(unsigned y=1; y<h; y++){

     for(unsigned x=1;x<w-1;x++){
        uint8_t acc=pSrc[y*w+x];
        acc=std::min(acc, pSrc[y*w+x-1]);
        acc=std::min(acc, pSrc[y*w+x+1]);
        acc=std::min(acc, pSrc[y*(w-1)+x]);
        acc=std::min(acc, pSrc[y*(w+1)+x]);
        pDst[y*w+x]=acc;
     }

  }
}
```

```
void erode_tbb(
  unsigned w, unsigned h,
  const uint8_t *pSrc,
  uint8_t *pDst
){
  tbb::parallel_for( 1u, h-1, [=](unsigned y){

     for(unsigned x=1;x<w-1;x++){
        uint8_t acc=pSrc[y*w+x];
        acc=std::min(acc, pSrc[y*w+x-1]);
        acc=std::min(acc, pSrc[y*w+x+1]);
        acc=std::min(acc, pSrc[y*(w-1)+x]);
        acc=std::min(acc, pSrc[y*(w+1)+x]);
        pDst[y*w+x]=acc;
     }

  });
}
```

Template Functions

```
#include <iostream>

template<class T>
void F(T x)
{
    std::cout<<x<<"\n";
}

int main(int, char *[])
{
    F(124.456);
    F("wibble");
    return 0;
}
```

- Template functions do not have a fixed input type

```cpp
template<class T>
void F(T &x, int n)
{
    x.Quack(n);
}

struct HowardTheDuck
{
    void Quack(int n)
    {
        for(int i=0;i<n;i++)
            std::cout<<"Quack";
    }
};

struct DaffyTheDuck
{
    void Quack(int n)
    {
        for(int i=0;i<n;i++)
            PlaySound("quack.wav");
    }
};
```

- Template functions do not have a fixed input type
- Templates rely on static “duck typing”
  - “*If it looks like a duck, and walks like a duck, and quacks like a duck, it is a duck*”
  - As long as the input type can do everything you want, it will compile

```cpp
int main(int, char *[])
{
    HowardTheDuck howard;
    DaffyTheDuck daffy;
    F(howard, 2); // prints "QuackQuack"
    F(daffy, 1);  // plays quack sound once
    return 0;
}
```

All about C++ lambdas

- Lambda functions are similar to matlab @ functions
  - Define pieces of code which can be treated as variables
  - Can capture parts of the environment in a closure

```
auto f = [](unsigned x){ return x*2; };        f=@(x)( x*2 );

unsigned x= f(4);                               x=f(4);
```

- C++ requires type annotations, as it is statically typed
  - The `auto` keyword means "*the type of the expression to the right*"
  - e.g.: `auto x=1.1;` means x has type `double`

Choosing your environment

- Lambdas can capture the environment in two ways
  - *Capture by value*: value of a variable fixed when lambda created
  - *Capture by reference*: variable is a reference to original

Capture by value

```
unsigned x=1;

auto f_val= [=](){ return x; };
auto f_ref= [&](){ return x; };

x=2;

std::cout<<f_val()<<"\n";  // prints 1
std::cout<<f_ref()<<"\n";  // prints 2
```

Capture by reference

Some tbb::parallel_for implementations

- It's useful to think about how parallel for can be built
- There are many possible ways to do parallel_for
  - Not all valid implementations are actually parallel
  - Not all parallel implementations are actually valid

Sequential implementation

- We are allowed to just do it sequentially
- Any correct TBB program should work with this version

```
namespace tbb{

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

}; // namespace tbb
```

Alternative sequential implementation

- The iterations can happen in any order
- **But**: must call each iteration once *and only* once

```
namespace tbb{

template<class TI, class TF>
void parallel_for(const TI &begin, const TI &end, const TF &f)
{
   TI i=end;
   do{
       i=i-1;
       f(i);
   }while(i!=begin);
}

}; // namespace tbb
```

Under the hood : std::thread

- `std::thread` is the C++ mechanism for creating threads
- Platform independent layer over low-level OS functions
- Designed to try to stop you doing anything too bad
  - But still very easy to have things go wrong: avoid if possible!

```
#include <thread> // Standard C++ header

void say_hello()
{ std::cout<<"Hello world"<<std::endl; }

int main(int argc, char *argv[])
{
   std::thread worker(say_hello); // Spawn new thread
   worker.join();  // Block till it finishes
   return 0;
}
```

Parallel implementation on std::thread

```
template<class TI, class TF>
void parallel_for(const TI &begin, const TI &end, const TF &f)
{
   std::vector<std::thread> threads;

   // Spin off threads for each iteration
   for(TI i=begin; i < end; i++){
       auto f_i = [&f,i](){  f(i);  } // Lambda to execute f(i)

       threads.push_back( std::thread( f_i ) ); // Start new thread
   }

   // Make sure all threads have finished
   for(unsigned i=0;i<threads.size();i++){
       threads[i].join();  // Wait until task has finished
   }
}
```

Scalability of std::thread version

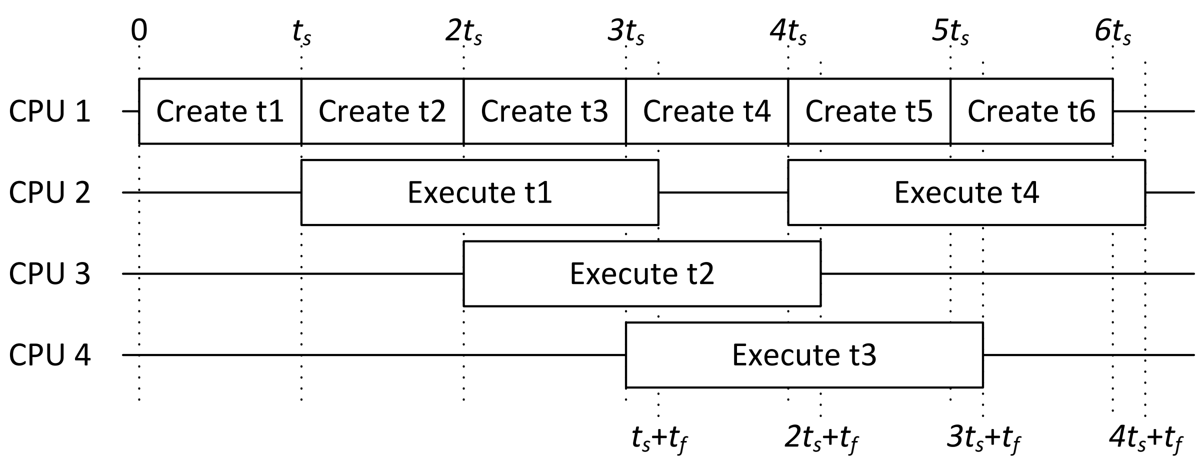
- Let us assume some simple timing behaviour
  - t_s : time taken to create and spawn one thread
  - t_f : time taken to execute the function f
- Also assume some system properties
  - OS is using time-slicing to assign threads to CPUs
  - There are P actual processor cores
- What is the expected behaviour for large `n=end-begin` ?

```
// Spin off threads for each iteration
for(TI i=begin; i < end; i++){
   auto f_i = [&f,i](){  f(i);  } // Lambda to execute f(i)

   threads.push_back( std::thread( f_i ) ); // Start new thread
}
```

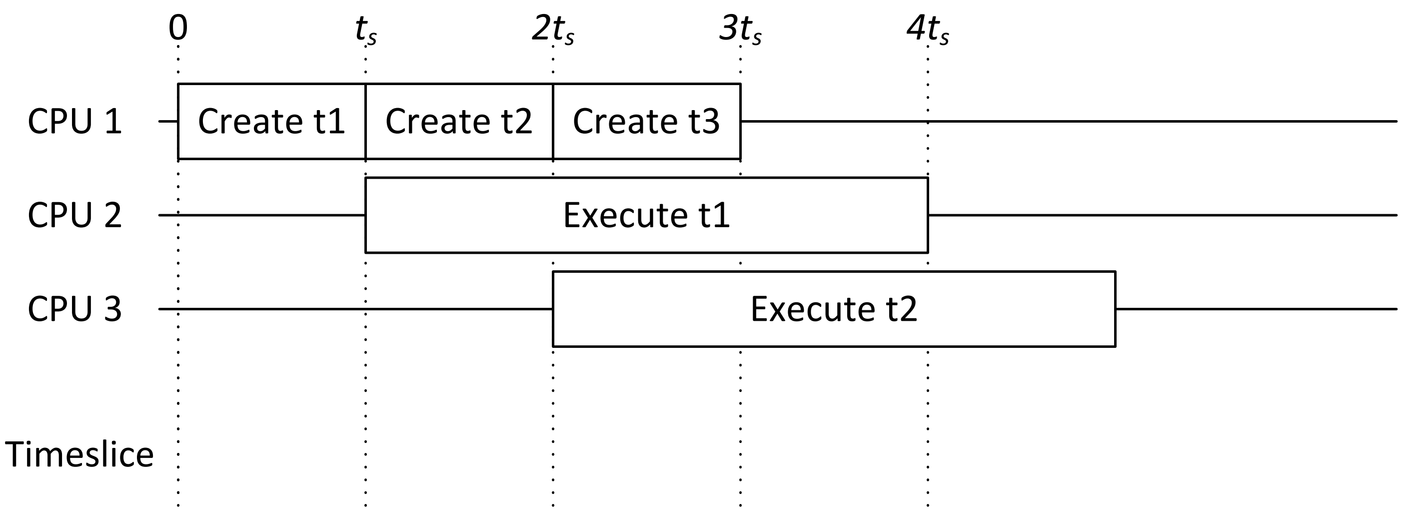
Under-utilised: $t_f / (P-1) < t_s$

- Each execution of *f* takes t_f time
  - All P processors can calculate P results in time t_f
  - **Or**: we can calculate a result about every t_f / P seconds
- One CPU constantly creates threads; others process them
  - 1 CPU : produces new thread every t_s seconds
  - P-1 CPUs : can complete one task every $t_f / (P-1)$ seconds
- Under-utilised when P-1 workers are faster than creator



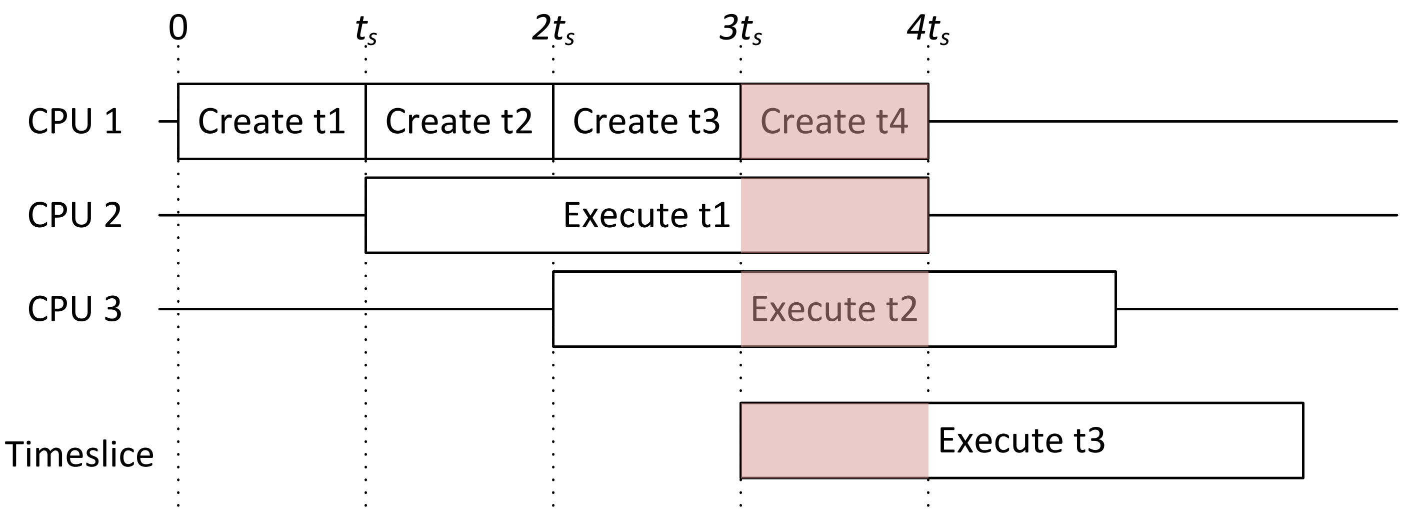
Overutilised : $t_f / (P\text{-}1) > t_s$

- Can now create threads faster than workers finish them
  - First *P-1* task are scheduled onto idle CPUs
  - *P*'th thread starts executing in addition to creator and P-1 threads



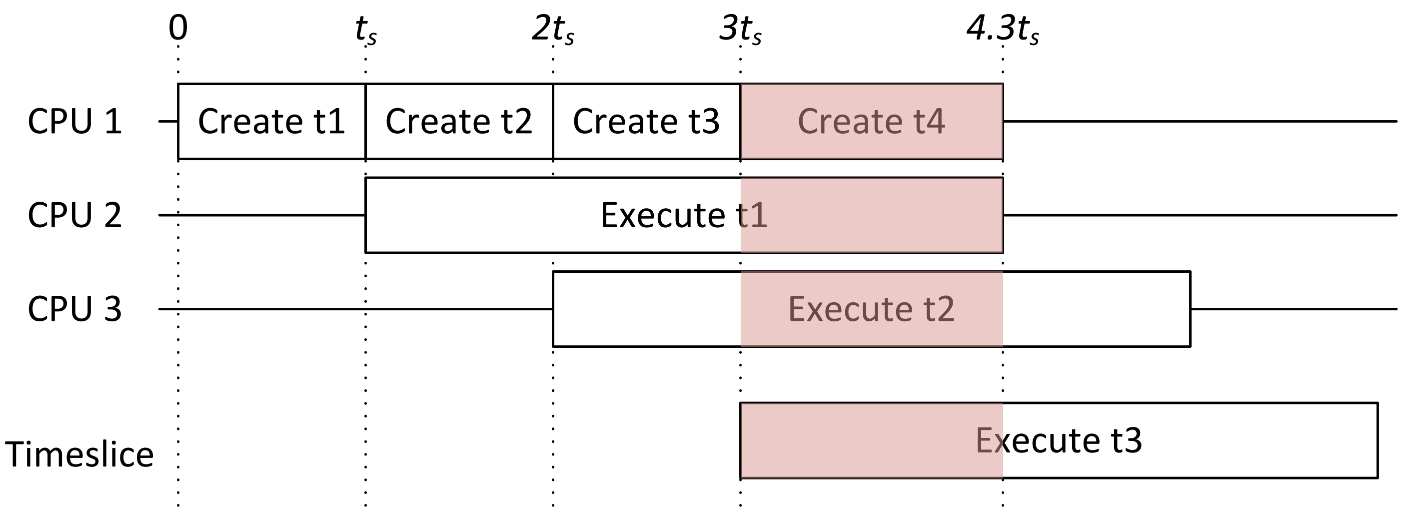
Overutilised : $t_f / (P\text{-}1) > t_s$

- Can now create threads faster than workers finish them
  - First *P-1* task are scheduled onto idle CPUs
  - *P*'th thread starts executing in addition to creator and P-1 threads
- OS will automatically start time-slicing
  - We have P+1 tasks, but just P processors: slow down by P/(P+1)



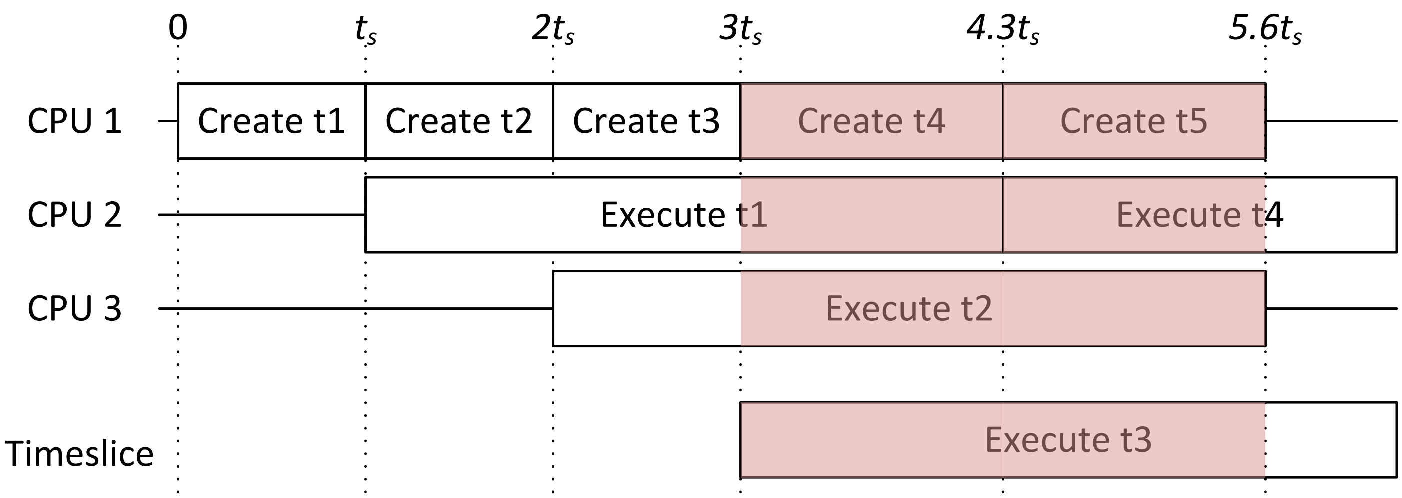
Overutilised : $t_f / (P-1) > t_s$

- Can now create threads faster than workers finish them
  - First *P-1* task are scheduled onto idle CPUs
  - *P*'th thread starts executing in addition to creator and P-1 threads
- OS will automatically start time-slicing
  - We have P+1 tasks, but just P processors: slow down by P/(P+1)



Overutilised : $t_f / (P\text{-}1) > t_s$

- Can now create threads faster than workers finish them
  - First *P-1* task are scheduled onto idle CPUs
  - *P*'th thread starts executing in addition to creator and P-1 threads
- OS will automatically start time-slicing
  - We have P+1 tasks, but just P processors: slow down by P/(P+1)



We are either under- or over-utilised

- *Underutilised*: limited by creation speed of work
  - Cannot exploit all the CPUs even though there is more work
- *Overutilised*: losing performance due to context switches
  - There is overhead when switching between OS threads
  - Each thread needs to warm up cache again
  - Increases memory pressure

- Worst case: continual slow-down
  - The cost of creating threads is partially borne by kernel
  - User code may slow down more than kernel code under load
  - Number of workers slowly goes up; completion rate goes down

Solving under-utilisation

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

        std::thread left(  // Spawn the left thread in parallel
            [&](){  parallel_for(begin, mid, f);    }
        );

        // Perform the right segment on our thread
        parallel_for(mid, end, f);

        // wait for the left to finish
        left.join();
    }
}
```

Creation of work using trees

- Tree starts on one thread

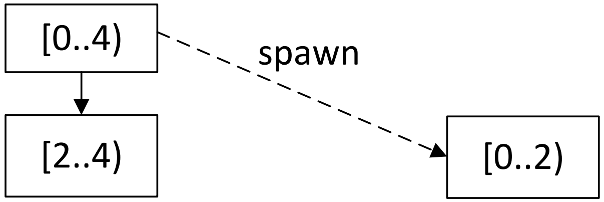
[0..4)

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

    std::thread left(
      [&](){  parallel_for(begin, mid, f);    }
    );
    parallel_for(mid, end, f);
    left.join();
  }
}
```

Creation of work using trees

- Tree starts on one thread
- Create thread to branch

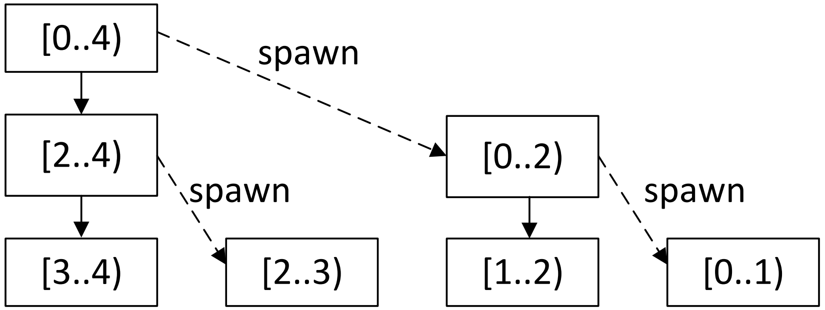



```
template<class TI, class TF>
void parallel_for(
     const TI &begin,const TI &end,
     const TF &f)
{
  if(begin+1 == end){
    f(begin);
  }else{
    TI mid=(begin+end)/2;

    std::thread left(
      [&](){  parallel_for(begin, mid, f);    }
    );
    parallel_for(mid, end, f);
    left.join();
  }
}
```

Creation of work using trees

- Tree starts on one thread
- Create thread to branch

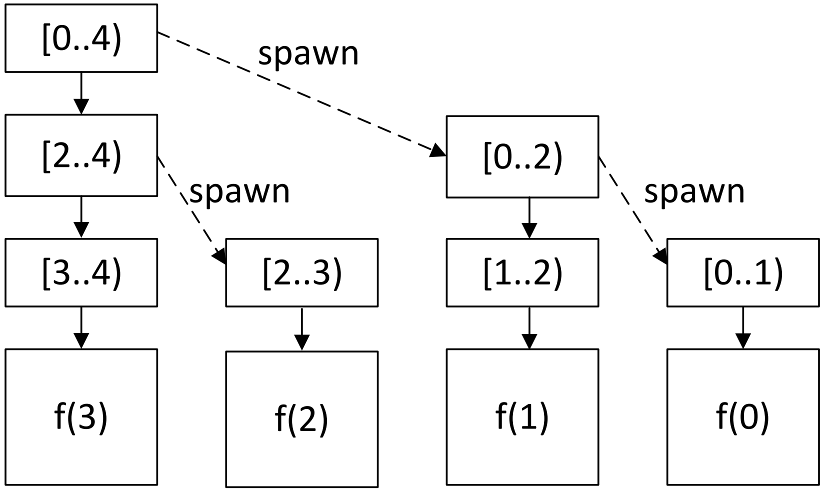



```
template<class TI, class TF>
void parallel_for(
     const TI &begin,const TI &end,
     const TF &f)
{
  if(begin+1 == end){
    f(begin);
  }else{
    TI mid=(begin+end)/2;

    std::thread left(
      [&](){  parallel_for(begin, mid, f);    }
    );
    parallel_for(mid, end, f);
    left.join();
  }
}
```

Creation of work using trees

- Tree starts on one thread
- Create thread to branch
- Execute function at leaves

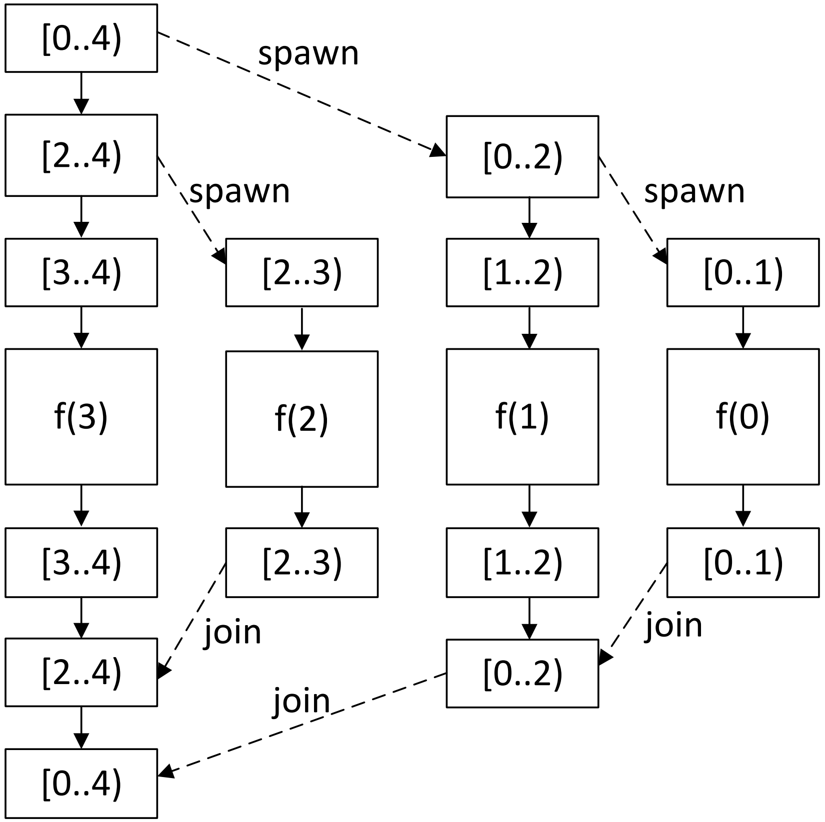



```
template<class TI, class TF>
void parallel_for(
     const TI &begin,const TI &end,
     const TF &f)
{
  if(begin+1 == end){
    f(begin);
  }else{
    TI mid=(begin+end)/2;

    std::thread left(
      [&](){  parallel_for(begin, mid, f);    }
    );
    parallel_for(mid, end, f);
    left.join();
  }
}
```

Creation of work using trees

- Tree starts on one thread
- Create thread to branch
- Execute function at leaves
- Join back up to the root




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

    std::thread left(
      [&](){  parallel_for(begin, mid, f);    }
    );
    parallel_for(mid, end, f);
    left.join();
  }
}
```

Properties of fork / join trees

- Recursively creating trees of work is very efficient
  - We are not limited to one thread creating all tasks
  - Exponential rather than linear growth of threads with time
- Problem solved?

- Growth of threads is **exponential** with time
- Can put significant pressure on the OS thread scheduler
  - Context switching 1000s of threads is very inefficient
- Each thread requires significant resources
  - Need kernel handles, stack, thread-info block, ...
  - Can't allocate more than a few thousand threads per process

Re-examining the goals

- What we want is parallel_for:

  “*Iterations* ***may*** *execute in parallel*”

- std::thread gives us something different:

  “*The new thread* ***will*** *execute in parallel*”

- Our thread based strategy is too eager to go parallel
- We want to go just parallel enough, then stay serial

Tasks versus threads

- A task is a chunk of work that can be executed
  - A task ***may*** execute in parallel with other tasks
  - A task will ***eventually*** be executed, but no guarantee on when
- Tasks are scheduled and executed by a run-time (TBB)
  - Maintain a list of tasks which are ready to run
  - Have one thread per CPU for running tasks
  - If a thread is idle, assign a task from the ready queue to it
  - No limit on number of tasks which are ready to run
  - (OS is still responsible for mapping threads to CPUs)
- TBB has a number of high-level ways to use tasks
  - But there is a single low-level underlying task primitive

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

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
   }
}
```

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