Threading Building Blocks

Introduction

The *Threading Building Blocks* (TBB) is a C++ library developed by Intel to specifically address programming of multi- and many-core systems. It supports Linux, Windows and MacOSX and all major C++ compilers.

TBB is available as a commercially supported library from Intel or as an open-source version from

http://threadingbuildingblocks.org

TBB provides algorithms and data structures to define *tasks* in a parallel program. These tasks are then mapped by an internal scheduler to worker threads.

In contrast to other thread programming packages the programmer has *no* access to these threads, only to the tasks.

TBB is using C++ templates extensively to minimise runtime overhead.

Remark

This course is based on Version 4 of TBB.

Introduction

Scheduling

By default tasks are enqueued into thread-local work queues.

In case of a load imbalance, *task-stealing* is used to transfer tasks to other threads.

Also, the scheduler tries to preserve *cache locality* by scheduling tasks first, which have been most recently in the cache.

This may result in *unfair* scheduling, i.e. other tasks may starve for processing resources.

Limitations

Due to unfair scheduling, TBB is *not* designed for

- I/O operations, where a task may block until data can be fetched or
- realtime operations, since no guarentee upon the execution time can be given.

Hello World

The standard introductory example looks as follows using TBB:

```
#include <iostream>
#include <tbb/tbb.h>                    // include all of TBB

using namespace tbb;                    // import TBB namespace

struct hello : public task {            // Hello World Task
  task * execute () {
    std::cout << "Hello, world!" << std::endl;
    return nullptr;
  }
};

int main () {
  hello & t = * new( task::allocate_root() ) hello;
  task::spawn_root_and_wait(t);
  return 0;
}
```

Here, printing “Hello, World” is defined as a task and handed to the TBB scheduler for execution.

Using the Intel compiler, a command line switch may be used to enable TBB:

```
> icpc -tbb -o hello hello.cc
```

All other compilers have to use the standard flags for include and library paths:

```
> g++ -I<path to TBB>/include -L<path to TBB>/lib -ltbb -o hello hello.cc
```

Library Initialisation

The TBB library needs no explicit initialisation. By default it will create worker threads for each processor (core).

To initialise the TBB library with an explicitly defined number of worker threads, the class *task_scheduler_init* is provided:

```
#include <tbb/task_scheduler_init.h>

void
main () {
  tbb::task_scheduler_init  tsi( nthreads );

  ...
}
```

Remark

The worker threads include the main thread, e.g. the initial thread of the process.

Lambda Functions

Lambda Functions

Lambda functions are *anonymous* functions in C++11, also called *closures*, which are especially useful when using TBB.

A lambda functions consists of a function body and a data environment, similar to a task.

The (almost) full definition of lambda functions is

```
[ capture ] ( params ) -> ret-type { body }
```

With

- capture: defining the data environment,
- params: function parameters,
- ret-type: defines the function return type and
- body: is the function body.

The last three correspond to the definition in standard named functions.

Captured Variables

Variables referenced in the function body have to be imported to the data environment of the lambda function.

Variables can either be captured by *copy* or by *reference*.

To capture a variable by copy, the variable has to be explicitly listed in the capture description:

```
[ x, y, i, j ] ...          // capture copy of x, y, i, j
```

Alternatively, with

```
[ = ] ...                   // all captured by copy
```

all variables referenced in the function body are automatically captured by copy.

If a variable should be captured by reference, the variable has to be prefixed by *&* in the capture description:

```
[ & x, & y, & i, & j ] ...  // capture reference of x, y, i, j
```

To capture all variables in the function body by reference, use

```
[ & ] ...                   // all captured by reference
```

Captured Variables

Capture by copy and by reference may also be combined:

```
[ x, & y, i, & j ] ...          // capture x, i by copy and y, j by reference
```

or

```
[ =, & y, & j ] ...             // capture all by copy except y and j
```

Remark

When using "=", it has to be first in the capture description.

An empty capture description will not capture any variable:

```
[ ] ...                         // do not capture any variable
```

Captured Variables

The exact value of a captured variable when executing the lambda function depends on how it is captured:

```
int    i = 1, j = 2;
double x = 3, y = 4;

auto f1 = [ x, y, i, j         ] () -> void { std::cout << i << j << x << y << std::endl; };
auto f2 = [ =                  ] () -> void { std::cout << i << j << x << y << std::endl; };
auto f3 = [ & x, & y, & i, & j ] () -> void { std::cout << i << j << x << y << std::endl; };
auto f4 = [ &                  ] () -> void { std::cout << i << j << x << y << std::endl; };
auto f5 = [ x, & y, i, & j     ] () -> void { std::cout << i << j << x << y << std::endl; };
auto f6 = [ =, & y, & j        ] () -> void { std::cout << i << j << x << y << std::endl; };

i = 5; y = 6;

f1(); f2(); f3(); f4(); f5(); f6();
```

The output is then:

```
1234
1234
5236
5236
1236
1236
```

Remark

Lambda functions have no explicitly nameable type. But it can be inferred by auto.

Simplified Lambda Functions

The full lambda definition can be simplified to

[*capture*] (*params*) { *body* }

if the body contains a *single* `return` statement, from which the return type can be automatically inferred:

```
auto f = [] ( const double x ) { return x*x; };
```

If no `return` statement exists, the return type defaults to `void`:

```
auto f = [] ( const double x ) { compute_something( x ); };
```

Similarly, if the lambda function has no parameters, it can be simplified to

[*capture*] -> *ret* { *body* }

e.g.

```
auto pi = [] -> double { return 355.0 / 113.0; };
```

Both simplifications can also be combined to

[*capture*] { *body* }

Loops

Simple Loops

The parallelisation of simple loops, e.g.

```
for ( size_t i = 0; i < n; ++i ) {
  ...
}
```

is provided by the TBB algorithm

```
template<typename index_t, typename func_t>
func_t parallel_for ( index_t         start,
                      index_t         end,
                      const func_t &  f );
```

Using `parallel_for`, the above loop translates into

```
parallel_for( 0, n, [] ( index_t i ) { ... } );
```

Alternatively, the loop body can be encapsulated by a named function:

```
void f ( index_t i ) {
  ...
}

parallel_for( 0, n, f );
```

Simple Loops

Example: Matrix Multiplication

Computing $C = A \cdot B$ with $A, B, C \in \mathbb{R}^{n \times n}$ in parallel using `parallel_for`:

```
parallel_for( 0, n,                                  // parallelise outer most loop
              [n,&A,&B,&C] ( const size_t i ) {      // capture A,B,C by reference
                for ( size_t j = 0; j < n; ++j ) {
                  double f = 0;
                  for ( size_t k = 0; k < n; ++k )
                    f += A(i,k)*B(k,j);
                  C(i,j) = f;
                }
              }
            );
```

The parallel speedup of the above implementation for $n = 1024$ is:

	Xeon X5650 (24 threads)	Xeon E5-2640 (24 threads)	XeonPhi 5110P (240 threads)
outer most loop only:	11.26x	11.29x	186.05x

Simple Loops

The `parallel_for` algorithm also accepts a *step* parameter:

```
template<typename index_t, typename func_t>
func_t parallel_for ( index_t         start,
                      index_t         end,
                      index_t         step,
                      const func_t &  f );
```

which enables the parallelisation of loops of the form

```
for ( size_t i = 0; i < n; i += step ) {
  ...
}
```

Simple Loops

The `parallel_for` algorithm also accepts a *step* parameter:

```
template<typename index_t, typename func_t>
func_t parallel_for ( index_t         start,
                      index_t         end,
                      index_t         step,
                      const func_t &  f );
```

which enables the parallelisation of loops of the form

```
for ( size_t i = 0; i < n; i += step ) {
  ...
}
```

Remark

The loop must not wrap around and the step value must be positive.

Simple Loops

Using the above form of `parallel_for`, each iteration of the loop induces a new task, i.e. a fine granular approach to loop parallelisation.

If the work per iteration is small, the management overhead may be too big for efficient parallelisation.

For such cases, the generalised version of `parallel_for` can be used, which uses *ranges* to specify the index set to operate on:

```
template< typename range_t, typename body_t >
func_t parallel_for ( const range_t & range,
                      const body_t &  body );
```

The *body* object may be a lambda function

```
[...] ( const range_t & r ) { ... }
```

or a standard C++ object with the following minimal interface:

```
struct body_t {
  body_t  ( const Body & );                          // copy constructor
  ~body_t ();                                        // destructor
  void operator () ( const range_t & r ) const;  // range based operator
};
```

Range Concept

TBB uses *ranges* to partition a given index set into sub sets for task definition. For this, it recursively subdivides the index set, until a indivisible set is reached. The constructed tasks are assigned to (or stolen from) the worker threads.

Each range in TBB has to implement a minimal interface to enable partitioning:

```
struct range_t {
  range_t  ( const range_t & )    // copy constructor
  range_t  ( range_t & r, split ) // split range into two sub ranges
  ~range_t ();                    // destructor

  bool empty        () const;     // return true if range is empty
  bool is_divisible () const;     // return true if range can be partitioned into
                                  // two sub ranges
};
```

The *splitting* constructor partitions the given range into two sub ranges, usually of equal size, and updates the range parameter with one of the new sub ranges:

```
struct int_range_t {
  int  lb, ub;              // lower and upper bound of interval [lb,ub)

  int_range_t::int_range_t ( int_range_t & r, split ) {
    int mid = (r.lb + r.ub) / 2;

    lb   = mid; ub = r.ub;  // this becomes second sub interval
    r.ub = mid;             // r    becomes first  sub interval
  }
};
```

Range Concept

TBB defines ranges for one, two and three dimensional indexsets.

```
template < typename value_t >      blocked_range;

template < typename row_value_t,
           typename col_value_t>   blocked_range2d;

template < typename page_value_t,
           typename row_value_t,
           typename col_value_t>   blocked_range3d;
```

For `value_t`, any integral or pointer datatype may be used. Furthermore, standard STL *random* iterators, e.g. for `std::vector`, are possible.

```
blocked_range< size_t >    r1( 0, 100 );                // index set [0,100)
blocked_range2d< size_t >  r2( 0, 100,  0, 10 );        // index set [0,100) x [0,10)
blocked_range3d< size_t >  r3( 0, 100,  0, 20,  0, 5 ); // index set [0,100) x [0,20) x [0,5)
```

Range iteration is performed in the standard STL way using *begin()* and *end()*. For the two and three dimensional ranges, the correspondinging sub ranges are accessed using *rows()* and *cols()* and, for `blocked3d_range` via *pages()*:

```
for ( auto i = r1.begin(); i != r1.end(); ++i ) ...

for ( auto i = r2.rows().begin(); i != r2.rows().end(); ++i )
  for ( auto j = r2.cols().begin(); j != r2.cols().end(); ++j )
    ...

for ( auto k = r3.pages().begin(); k != r3.pages().end(); ++k )
  for ( auto i = r3.rows().begin(); i != r3.rows().end(); ++i )
    for ( auto j = r3.cols().begin(); j != r3.cols().end(); ++j )
      ...
```

Range Concept

Using ranges, the parallel loop

```
parallel_for( 0, n, [] ( index_t i ) { ... } );
```

can be rewritten as

```
parallel_for( blocked_range< size_t >( 0, n ),
              [] ( const blocked_range< size_t > & r ) {
                for ( auto i = r.begin(); i != r.end(); ++i ) {
                  ...
              } }
            );
```

Range Concept

Using ranges, the parallel loop

```
parallel_for( 0, n, [] ( index_t i ) { ... } );
```

can be rewritten as

```
parallel_for( blocked_range< size_t >( 0, n ),
              [] ( const blocked_range< size_t > & r ) {
                for ( auto i = r.begin(); i != r.end(); ++i ) {
                  ...
              } }
            );
```

Example: Matrix Multiplication

With `blocked_range2d`, both outer loops can be parallelised:

```
parallel_for( blocked_range2d< size>t( 0, n,  0, n ),
              [n,&A,&B,&C] ( const blocked_range2d< size_t > & r ) {
                for ( auto i = r.rows().begin(); i < r.rows().end(); ++i ) {
                  for ( auto j = r.cols().begin(); j < r.cols().end(); ++j ) {
                    double f = 0;
                    for ( size_t k = 0; k < n; ++k )
                      f += A(i,k)*B(k,j);
                    C(i,j) = f;
              } } } );
```

This yields a speedup of:

	Xeon X5650 (24 threads)	Xeon E5-2640 (24 threads)	XeonPhi 5110P (240 threads)
both outer loops:	11.20x	11.29x	211.88x

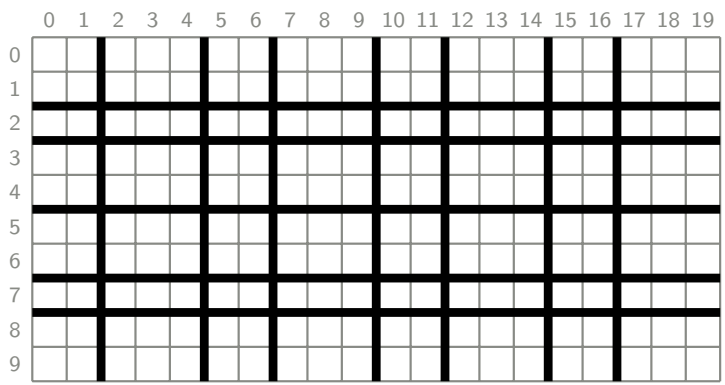
Range Concept

Beside the lower and upper bound for each dimension, an optional *grain size* may be specified, which defines the minimal chunk size for recursive partitioning, which is *one* by default.

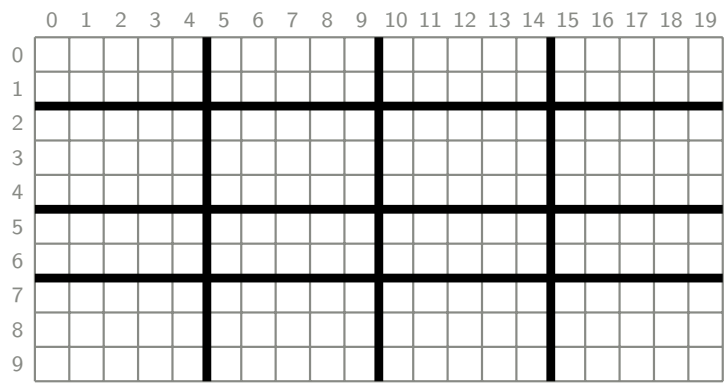
```
blocked_range   ( value_t      lb, value_t      ub, size_t      grainsize );

blocked_range2d ( value_t  row_lb, value_t  row_ub, size_t  row_grainsize,
                  value_t  col_lb, value_t  col_ub, size_t  col_grainsize );

blocked_range3d ( value_t page_lb, value_t page_ub, size_t page_grainsize,
                  value_t  row_lb, value_t  row_ub, size_t  row_grainsize,
                  value_t  col_lb, value_t  col_ub, size_t  col_grainsize );
```

In the following example, the task borders for different grain sizes for a two dimensional index set are shown. Hereby, the uneven one dimensional range sizes are a result of the recursive sub division.

```
blocked_range2d< size_t >  r1( 0, 10, 2
                               0, 20  3 );
```

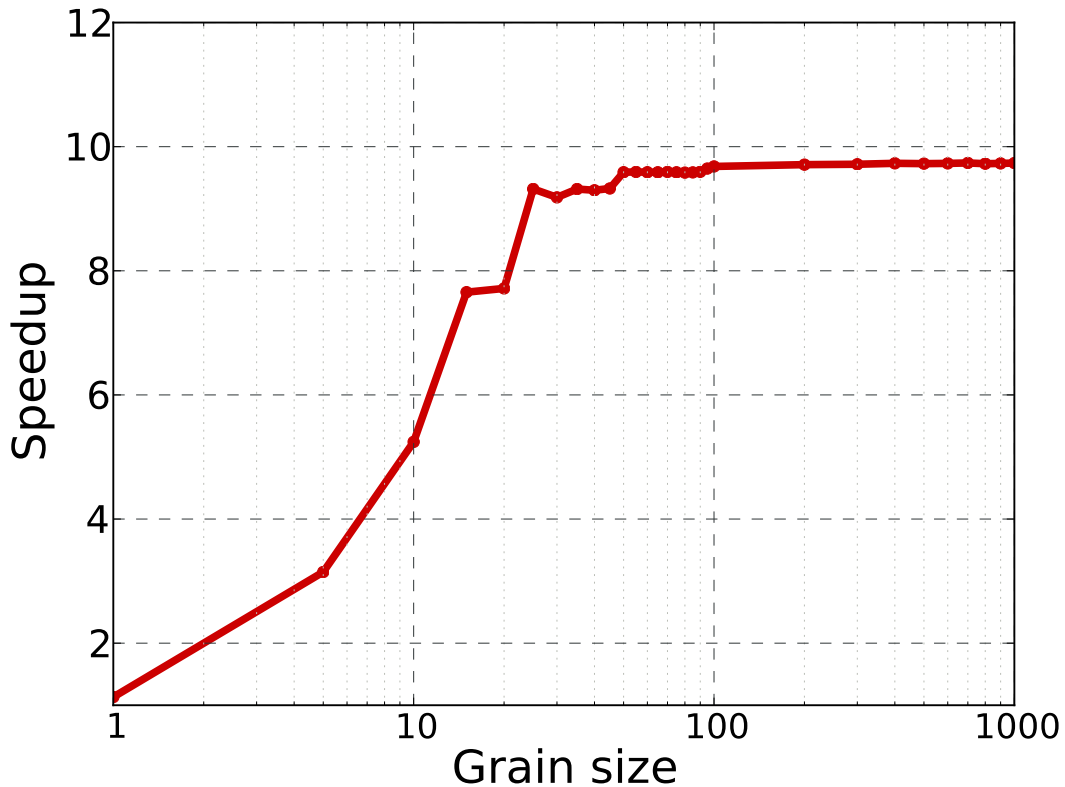



```
blocked_range2d< size_t >  r2( 0, 10, 4
                               0, 20, 6 );
```



Range Concept

For the computation $x_i = x_i + \sqrt{y_i}$ with $x, y \in \mathbb{R}^n$, the parallel speedup depending on the grain size on a 2-CPU Xeon E5-2640 is presented in the following diagram:



For a very small grain size, management overhead clearly dominates the runtime, preventing any speedup.

Partitioners

Ranges define a partition of an index set, e.g. into chunks of at most grain size.

However, a parallel loop starts with the whole index set and successively divides it into smaller sub index sets, such that each thread will have some tasks to work on.

This process is controlled by *partitioners* in TBB, of which the following are provided:

`auto_partitioner`:	divide range into sub ranges until all threads have (almost) equal load,
`affinity_partitioner`:	similar to `auto_partitioner` but tries to maximise cache locality,
`simple_partitioner`:	divides range until grain size is reached.

The partitioner is an optional argument to `parallel_for`:

```
parallel_for ( start, end, func, partitioner );
parallel_for ( range, body, partitioner );
```

e.g.

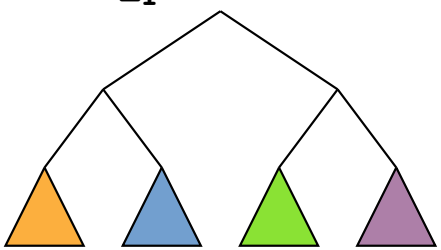
```
parallel_for( blocked_range< size_t >( 0, n ),
              [] ( const blocked_range< size_t > & r ) { ... },
              simple_partitioner()
            );
```

Partitioners

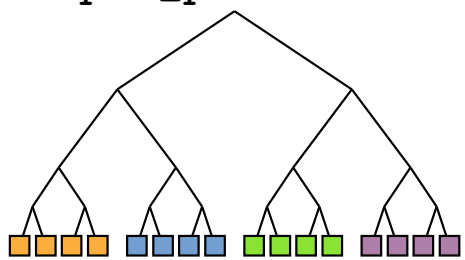
By default, `auto_partitioner` is used. Since the sub division is stopped, when all threads have enough work, the specified grain size is not neccessarily reached. Therefore, even if the grain size is one, the efficiency will often *not* deteriorate.

In contrast to this, the `simple_partitioner` will always split ranges until the the sub ranges are not larger than the grain size, hence, the latter needs to be chosen *appropriately*.

`auto_partitioner`



`simple_partitioner`



The `affinity_partitioner` may increase performance if several iterations are performed for the *same* data set and the per task data *fits* in the local cache.

Reductions

Parallel reduction operations are provided in TBB by the *parallel_reduce* algorithm:

```
template< typename range_t, typename value_t,
          typename func_t,  typename reduce_t >
value_t parallel_reduce ( const range_t &  range,
                          const value_t &  identity,
                          const func_t &   func,
                          const reduce_t & reduce );
```

Here, `func` is the function accumulating values over a given range. It has to implement

```
struct func_t {
  value_t operator () ( const range_t & r, const value_t & val );
};
```

where `val` specifies a start value for the local reduction.

The combination of locally computed values is implemented by `reduce_t`:

```
struct reduce_t {
  value_t operator () ( const value_t & x1, const value_t & x2 );
};
```

Finally, `identity` denotes the identity with respect to the reduction operation.

Reductions

For a standard reduction based on addition, the corresponding `func_t` implementation could be

```
struct arraysum_t {
  std::vector< double > & x;                              // array of coefficients to reduce

  arraysum_t ( std::vector< double > & ax ) : x( ax ) {}  // constructor

  double operator () ( const blocked_range< size_t > & r, const double val ) {
    double  s = val;
    for ( auto i = r.begin(); i != r.end(); ++i )
      s += x[i];
    return s;
  }
};
```

Similarly, for `reduce_t`, one could use

```
struct plus_t {
  double operator () ( const double  x1, const double  x2 ) const {
    return x1+x2;
  }
};
```

Finally, the parallel reduction looks as

```
sum = parallel_reduce( blocked_range< size_t >( 0, n, OPT_GRAIN_SIZE ),
                       double(0),
                       arraysum_t( x ),
                       plus_t() );
```

Reductions

The main advantage of TBB reductions over other implementations, e.g. OpenMP, is that it is *not* restricted to elementary data types.

As an example, if the scalar values in the previous summation are replaced by vectors in $\mathbb{R}^3$, the parallel reduction based on lambda functions is:

```
std::vector< Vector3 >  vectors( n );
...
sum = parallel_reduce( blocked_range< size_t >( 0, n, OPT_GRAIN_SIZE ),

                       Vector3(0,0,0),

                       [&vectors]( const blocked_range< size_t > & r,
                                   const Vector3 &                 val ) {
                         Vector3  v = val;
                         for ( auto i = r.begin(); i != r.end(); ++i )
                           v += vectors[i];
                         return v;
                       },

                       std::plus< Vector3 >()
                     );
```

All that is needed here are operators `+=` and `+` for objects of type `Vector3`.

Remark

The STL provides a wide variety of binary operations in the `functional` module, e.g. `plus`, `minus`, `multiplies`, `divides`, etc..

Reductions

Extending the previous example to matrices would result in many copy operations, since each local reduction or combination would return a matrix object.

To avoid such copies, TBB provides an imperative form of parallel reductions:

```
template< typename range_t, typename body_t >
value_t parallel_reduce ( const range_t & range,
                          const body_t &  body );
```

Here, `body_t` must implement the following interface:

```
struct body_t {
  body_t           ( body_t & b, split );      // splitting constructor
  ~body_t          ();                         // destructor
  void operator () ( const range_t & r );      // sub range accumulation
  void join        ( const body_t &  b );      // combine parameter with local result
};
```

The splitting constructor is used by TBB to make copies of the `body_t` object for different worker threads.

It may be executed concurrently with the sub range accumulation or the `join` function. Hence, special protection for object local variables may be neccessary, allthough this is usually not needed.

Reductions

Using the imperative form of `parallel_reduce`, the matrix sum $\sum_i M_i$ with $M_i \in \mathbb{R}^{m \times m}, 0 \leq i < n$, is computed via:

```
struct MatrixSum {
  std::vector< Matrix > &  matrices;
  Matrix                   loc_sum;

  MatrixSum ( MatrixSum &  ms, split )
    : matrices( ms.matrices ), loc_sum( ZERO_MATRIX )
  {}

  void operator () ( const blocked_range< size_t > & r ) {
    Matrix  M( ZERO_MATRIX );

    for ( auto i = r.begin(); i != r.end(); ++i )
      M += matrices[i];

    loc_sum += M;
  }

  void join ( MatrixSum & M ) {
    loc_sum += M.loc_sum;
  }
};
```

Here, no additional copy is performed, since only local variables are updated and *not* returned from the involved functions.

The actual parallel reduction then looks as

```
std::vector< Matrix >  matrices( n );
...
MatrixSum  sum( matrices );
parallel_reduce( blocked_range< size_t >( 0, n, OPT_GRAIN_SIZE, sum );
```

Reductions

Example: Matrix Multiplication

Parallelisation of $C = A \cdot B$ may be extended to the innermost loop, where a reduction is performed for the dot product:

```
parallel_for(
  blocked_range2d< size_t >( 0, n, OPT_GRAIN_SIZE,
                             0, n, OPT_GRAIN_SIZE ),
  [n,&A,&B,&C] ( const blocked_range2d< size_t > & r ) {
    for ( auto i = r.rows().begin(); i < r.rows().end(); ++i ) {
      for ( auto j = r.cols().begin(); j < r.cols().end(); ++j ) {
        C(i,j) = parallel_reduce( blocked_range< size_t >( 0, n, OPT_GRAIN_SIZE ),
                                  double(0),
                                  [i,j,&A,&B,&C]( const blocked_range< size_t > & ri,
                                                  double val ) {
                                    double  f = val;
                                    for ( auto k = ri.begin(); k != ri.end(); ++k )
                                      f += A(i,k)*B(k,j);
                                    return f;
                                  },
                                  std::plus< double >() );
  } } } );
```

Comparing the previous approaches for $n = 1024$ and a grain size of 64 we have:

	Xeon X5650 (24 threads)	Xeon E5-2640 (24 threads)	XeonPhi 5110P (240 threads)
outer most loop only:	10.41x	9.80x	16.16x
both outer loops:	11.31x	11.00x	121.47x
outer loops plus reduction:	10.08x	10.25x	167.39x

Deterministic Reductions

By default, the splits and joins of parallel reduction operations may be performed in any order. Since for the reduction itself, associativity is assumed, this poses no problem *in theory*.

In practise however, the results of parallel reductions may differ between different runs.

The enforce the same order of splits and joins, TBB provides a deterministic variant of `parallel_reduce`:

```
template< typename range_t, typename value_t,
          typename func_t,  typename reduce_t >
value_t parallel_deterministic_reduce ( const range_t &  range,
                                        const value_t &  identity,
                                        const func_t &   func,
                                        const reduce_t & reduce );

template< typename range_t, typename body_t >
value_t parallel_deterministic_reduce ( const range_t & range,
                                        const body_t &  body );
```

The definition of the arguments is identical to the standard parallel reduction.

Deterministic Reductions

It is important to note, that the deterministic computation does *not* reproduce sequential computation. As an example, computing

$$\sum_{i=1}^{n} \frac{(-1)^{i+1}}{i}$$

for $n = 10000000$ sequentially via

```
for ( auto v : x ) sum += v;
```

gives

```
6.9314713055990318e-01
```

If the same reduction is computed with

```
parallel_deterministic_reduce( blocked_range< size_t >( 0, n ), double(0),
                               [&x]( const blocked_range< size_t > & r, double s ) { ... }
                               std::plus< double >() );
```

the output is

```
6.9314713055994759e-01
```

Furthermore, the output is dependent on the grain size.

General Loops

For all previous loops the loop index set was known before parallelisation. Furthermore, a random access was implicitly required to access elements of individual tasks, e.g. vector coefficients to work on.

If this is not the case, `parallel_for` can not be used for loop parallelisation.

Instead, TBB provides the *`parallel_do`* algorithm:

```
template < typename iterator, typename body_t >
void parallel_do ( iterator first,
                   iterator last,
                   body_t     body );
```

The iterator type for `parallel_do` only has to provide a minimal function set, e.g. construction and increment. Hence, even `std::list` may be used.

If the iteration space is fixed, e.g. but still restricted to `std::list`, `body_t` has to provide only evaluation per set item:

```
struct body_t {
  void operator () ( [const] item_t & item ) const;  // "const" depends on iterator
};
```

General Loops

Fetching new tasks from the iterator in `parallel_do` is considered a critical section, hence performed strictly sequential.

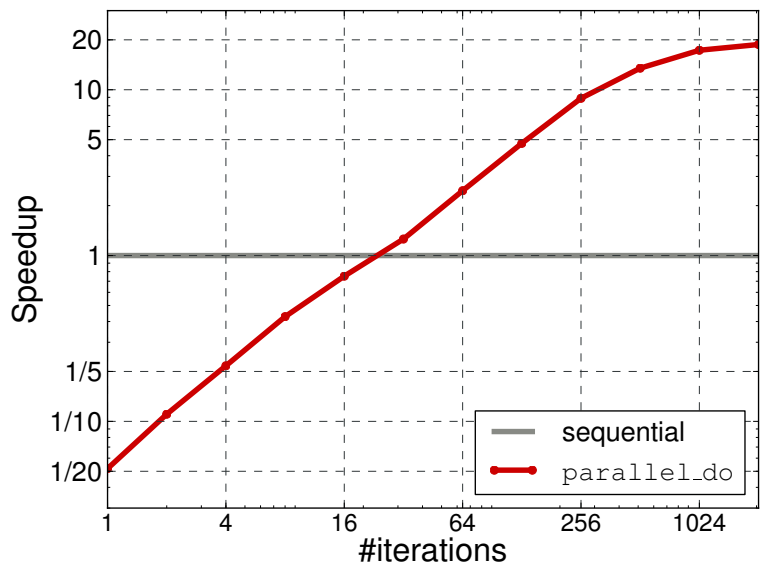
Therefore, the tasks should be sufficiently large to not let management overhead dominate.

As an example, for a list of `double` values the following computation was performed on a 2-CPU Xeon E5-2640 with an increasing number of iteration:

```
parallel_do( x.begin(), x.end(), []( double & v ) {
  for ( int it = 0; it < N_ITER; ++it )
    v = std::sin( v );
} );
```

The speedup depending on the per task work is show in the left diagram.

Especially for tasks with little work, `parallel_do` is *not* usable for an efficient parallelisation.



General Loops

Fetching new tasks from the iterator in `parallel_do` is considered a critical section, hence performed strictly sequential.

Therefore, the tasks should be sufficiently large to not let management overhead dominate.

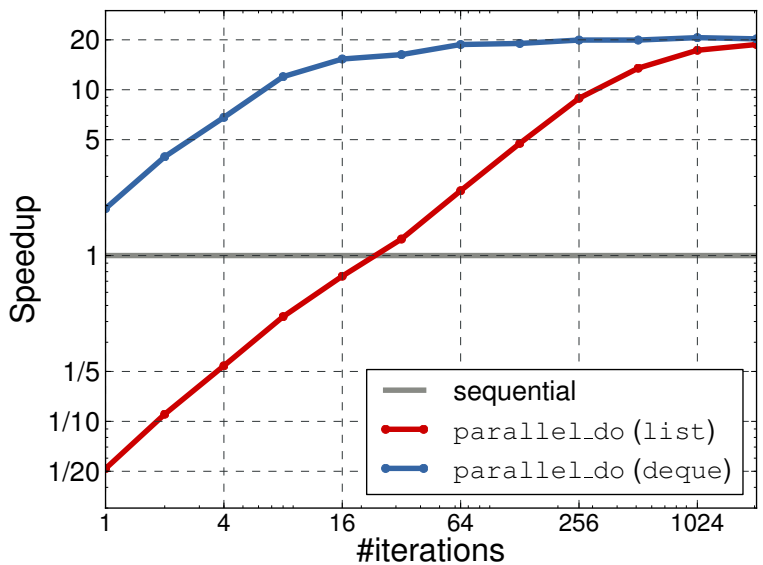
As an example, for a list of `double` values the following computation was performed on a 2-CPU Xeon E5-2640 with an increasing number of iteration:

```
parallel_do( x.begin(), x.end(), []( double & v ) {
  for ( int it = 0; it < N_ITER; ++it )
    v = std::sin( v );
} );
```

The speedup depending on the per task work is show in the left diagram.

Especially for tasks with little work, `parallel_do` is *not* usable for an efficient parallelisation.

The main reason for the inefficiency is the iterator of `std::list`. Using random iterators, the management overhead is much smaller.



General Loops

Fetching new tasks from the iterator in `parallel_do` is considered a critical section, hence performed strictly sequential.

Therefore, the tasks should be sufficiently large to not let management overhead dominate.

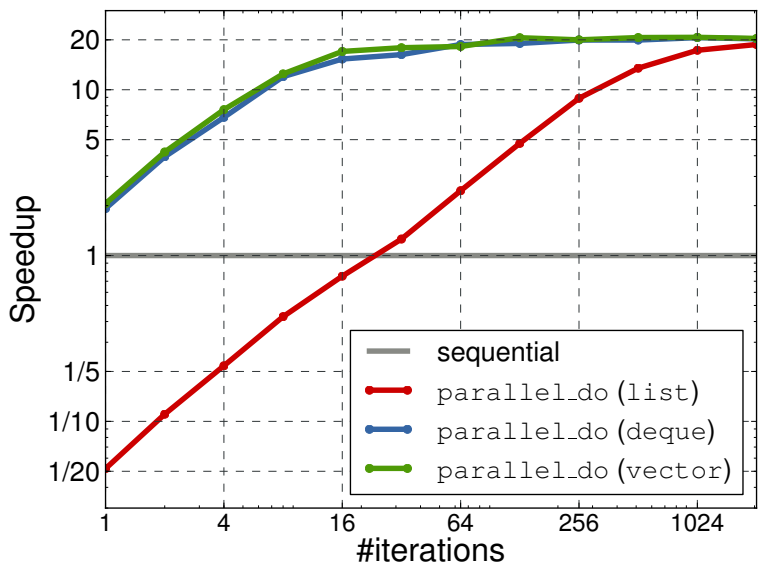
As an example, for a list of `double` values the following computation was performed on a 2-CPU Xeon E5-2640 with an increasing number of iteration:

```
parallel_do( x.begin(), x.end(), []( double & v ) {
  for ( int it = 0; it < N_ITER; ++it )
    v = std::sin( v );
} );
```

The speedup depending on the per task work is show in the left diagram.

Especially for tasks with little work, `parallel_do` is *not* usable for an efficient parallelisation.

The main reason for the inefficiency is the iterator of `std::list`. Using random iterators, the management overhead is much smaller.



General Loops

The `parallel_do` algorithm also permits new tasks to be added to the task set *during* execution.

For this, the evaluation function of the `body_t` object has to accept an additional argument:

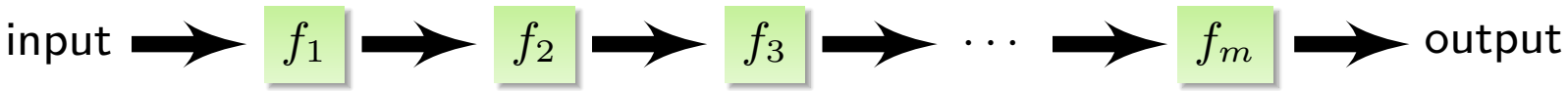
```
struct body_t {
  void operator () ( [const] item_t &                item,
                     parallel_do_feeder< item_t > & feeder ) const;
};
```

The `parallel_do_feeder` object provides a function *`add`* to extend the task set:

```
parallel_do( x.begin(), x.end(),
             []( item_t &                        item,
                 parallel_do_feeder< item_t > & feeder ) {
               handle( item );

               if ( more_work_available() )
                 feeder.add( more_work() );
             } );
```

Pipeline

For a set of input items, to which functions $f_1, \ldots, f_m$ are successively applied, e.g. $f_m(\ldots f_3(f_2(f_1(\cdot))))$, the pipeline algorithm model may be used:



This model is provided by TBB by the *parallel_pipeline* algorithm, while each stage in the pipeline corresponds to a *filter*:

```
void
parallel_pipeline ( size_t                                max_parallel,
                    const filter_t< void, void > & filter_chain );
```

The filter chain itself consists of concatenated filters and can be constructed using *make_filter*:

```
make_filter< void,     outinp_t  >( modei, input  )   &  // feed pipeline
make_filter< input1_t, output1_t >( mode1, func1  )   &  // apply f_1
make_filter< input2_t, output2_t >( mode2, func2  )   &  // apply f_2
...
make_filter< inputm_t, outputm_t >( modem, funcm  )   &  // apply f_m
make_filter< inpout_t, void      >( modeo, output ) );   // take items out
```

The output type of filter j must be identical to the input type of filter $j+1$. The input type of the first and the output type of the last filter have to be *void*.

Pipeline

The *mode* of a filter may be one of

`filter::serial_in_order`	applies function to items one at a time and always in the same order,
`filter::serial_out_of_order`	applies function to items one at a time in no particular order,
`filter::parallel`	applies function to several items in parallel.

The maximal degree of concurrency for a filter is defined by the value of `max_parallel`. It prevents parallel filters to accumulate an arbitrary number of items, if a following serial filter can not handle the input stream fast enough.

Except for the first function, all function objects *func* have to implement

```
output_t  func::operator ()  ( input_t  item );
```

with `output_t` and `input_t` being the input and output types of the corresponding filter.

Pipeline

As the first function object feeds the pipeline, it also controls the end of the pipeline via a *flow_control* object:

```
outinp_t  func::operator  ( flow_control & fc );
```

If all input items are processed, *fc.stop()* has to be called. The return value of the function is afterwards *not* used:

```
output0_t func::operator ( flow_control & fc ) {
  if ( input.empty() ) {
    fc.stop();                  // signal pipeline stop
    return output0_t(0);        // return dummy value
  } else {
    return next( input );
} }
```

Pipeline

For the introductory setting, the full source using lambda functions is:

```
parallel_pipeline(
  max_parallel,

  // insert items into pipeline
  make_filter< void, double >(
    filter::serial_in_order, [&input] ( flow_control & fc ) {
      if ( input.empty() ) {
        fc.stop();
        return 0.0;
      } else {
        double v = * (input.begin());
        input.pop_front();
        return v;
    } }
  )
  &
  // apply f1 ... fm in parallel
  make_filter< double, double >( filter::parallel, [] ( double v ) { return f1( v ); } )
  &
  make_filter< double, double >( filter::parallel, [] ( double v ) { return f2( v ); } )
  &
  make_filter< double, double >( filter::parallel, [] ( double v ) { return f3( v ); } )
  &
  ...
  &
  make_filter< double, double >( filter::parallel, [] ( double v ) { return fm( v ); } )
  &
  // finish pipeline
  make_filter< double, void >(
    filter::serial_in_order, [&output] ( const double v ) {
      output.push_back( v );
    }
  )
);
```

Pipeline

Filters may also be created in advance using the *filter_t* class:

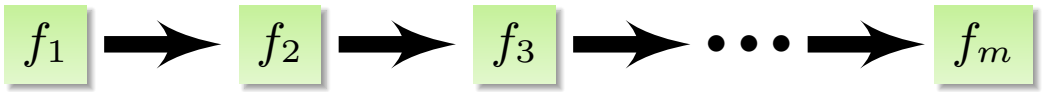
```
filter_t< void, double   > fin(  filter::serial_in_order,
                                 [&input] ( flow_control & fc ) {
                                   if ( input.empty() ) {
                                     fc.stop();
                                     return 0.0;
                                   } else {
                                     double v = * (input.begin());
                                     input.pop_front();
                                     return v;
                                 } } );
filter_t< double, double > ff1(  filter::parallel, [] ( double v ) { return f1( v ); } );
filter_t< double, double > ff2(  filter::parallel, [] ( double v ) { return f2( v ); } );
filter_t< double, double > ff3(  filter::parallel, [] ( double v ) { return f3( v ); } );
filter_t< double, double > ff4(  filter::parallel, [] ( double v ) { return f4( v ); } );
filter_t< double, void   > fout( filter::serial_in_order,
                                 [&output] ( const double v ) {
                                   output.push_back( v );
                                 } );
filter_t< void, void >     f = fin & ff1 & ff2 & ff3 & ff4 & fout;  // create pipeline

parallel_pipeline( max_parallel, f );
```

This also allows to *reuse* filters in a pipeline:

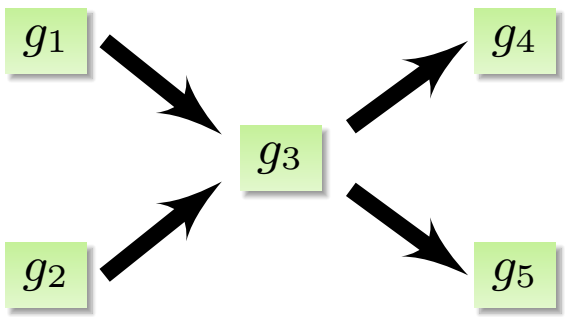
```
filter_t< void, void >     f = fin & ff1 & ff2 & ff3 & ff4 & ff4 & fout;
```

Non-Linear Pipelines

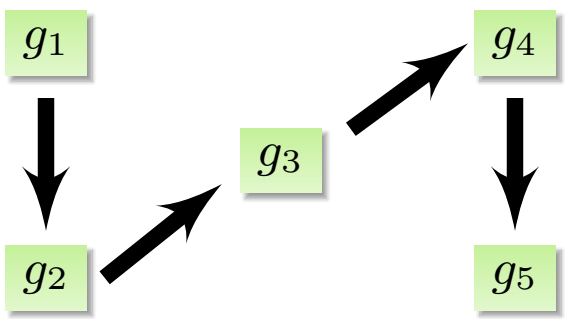
Pipelines in TBB have to be linear, e.g.



Multiple inputs or outputs are *not* supported, e.g.



Instead, such pipelines have to be *linearised*:



Pipeline Throughput

The rate at which items are processed by a pipeline depends on several parameters:

- the value of `max_parallel`,
- slowest filter,
- granularity.

Choosing `max_parallel` too small may prohibit using all available parallel resources. On the other hand, a value too large may induce too much overhead, e.g. memory.

In any case, the filter with the least throughput rate will determine the throughput of the whole pipeline. This especially applies to serial or IO-bound filter.

Finally, having many filters in the pipeline executing only little work will create more management overhead. Hence, each function should have a reasonable runtime for efficient parallelisation.

Pipeline Class

Beside the `parallel_pipeline` algorithm, also a *`pipeline`* class exists in TBB:

```
class pipeline {
public:
  void add_filter ( filter &  f );          // add filter to pipeline
  void run        ( size_t max_parallel ); // execute pipeline
  void clear();                            // remove all filters
};
```

However, in contrast to the `filter_t` class, `filter` is *not* type safe. New filters have to be derived from `filter` and need to overload the `()` operator:

```
class my_filter : public filter {
  void *  operator ()  ( void * item );
};
```

Input items have to be cast to the actual type, while ensuring that all items have a lifetime equal to the pipeline itself.

The end of the input stream is signaled by a `NULL` return value of the first filter and the output of the last filter is ignored.

Task Groups

Task Groups

TBB provides a high-level interface to tasks, which allows task definition by using standard C++ function objects.

Each of these tasks belongs to a *task group* implemented by the class *task_group*:

```
class task_group {
  template< typename func_t > void run          ( const func_t & f );
  task_group_status           void wait         ();
  template< typename func_t > void run_and_wait ( const func_t & f );
};
```

Spawning a task in a task group is performed using the function `run()`, whereas `wait()` will block until all tasks in the task group have finished execution:

```
task_group  g;

g.run( [] { ... } );
g.run( [] { ... } );

g.wait();
```

Spawning the last task in a group and waiting for all tasks is combined in `run_and_wait()`.

Task Groups

Beside spawning and waiting, tasks in a task group may be *cancelled* via

```
void task_group::cancel ();
```

Tasks, which are not yet running will *not* be scheduled for execution if the group has been cancelled. However, if a task is already running, cancellation has *no* effect.

The return value of the function `wait()` gives information about the completion and cancellation status of the task group using the `task_group_status` enum:

- `not_complete`: group was not cancelled and there are still tasks which have not completed,
- `complete`: group was not cancelled and all tasks completed,
- `canceled`: group was cancelled.

The current cancellation status can also be retrieved using the function

```
void task_group::is_canceling ();
```

Remark

If an exception is thrown within a task, the task group is cancelled.

Task Groups

Example: Dot Product

For computing the dot product $\langle x, y \rangle$ with $x, y \in \mathbb{R}^n$, a recursive algorithm is chosen. Each recursive call corresponds to a new task, belonging to a new task group:

```
double parallel_dot ( const std::vector< double > & x,
                      const std::vector< double > & y,
                      const size_t lb, const size_t ub ) {
  if ( ub - lb < MIN_SIZE ) {
    return std::inner_product( & x[lb], & x[ub], & y[lb], 0.0 );
  } else {
    task_group  g;
    double      res1 = 0;
    double      res2 = 0;

    g.run( [ub,lb,&res1,&x,&y] { res1 = parallel_dot( x, y, lb, (ub-lb)/2 ); } );
    g.run( [ub,lb,&res2,&x,&y] { res2 = parallel_dot( x, y, (ub-lb)/2, ub ); } );
    g.wait();

    return res1 + res2;
  }
}
```

Tasks

The `task` Class

The fundamental building block of the low-level task based programming in TBB is the *`task`* class and its *`execute`* function:

```
class task {
  public:
    virtual task * execute() = 0;
    ...
};
```

All user tasks have to be derived from `task` and overload `execute`.

Each task has an optional *parent* and a *reference counter* associated with it, accessible via

```
task * task::parent     () const;
void   task::set_parent ( task * p );
```

```
void task::set_ref_count       ( int count );
void task::increment_ref_count ();
int  task::decrement_ref_count ();
```

The reference counter of a task t contains the number of tasks to which t is the parent. Modifications of a reference counter are always atomic.

A *root* task has no parent and a reference counter of 0. Furthermore, if the `parent` field of a task is set, this task is a *child* task of the corresponding parent.

Allocation

Tasks in TBB provide special allocation routines for an efficient management of tasks.

```
static proxy1_t allocate_root                ();
proxy2_t        allocate_child               ();
static proxy3_t allocate_additional_child_of ( task & t );
```

Remark

The allocation function return internal proxy objects, which handle the actual allocation.

Depending on the allocation routine, the parent and the reference counter of the new (or calling) task is set:

	parent	ref. counter
`allocate_root()`	`nullptr`	0
`t->allocate_child()`	`t`	0
`allocate_additional_child_of( t )`	`t`	0, `t.refcount++`

All tasks must be allocated by these functions. Otherwise, the results of the allocation is undefined.

Synchronisation

TBB provides several methods to spawn new tasks and synchronise with the end of tasks, depending on how the tasks were allocated.

For a parent task p, a child task t is put into the thread-local work set via

```
static void task::spawn ( task & t );
```

For further synchronisation to work, the value of the reference counter of p is important. *Before* spawning any task, it has to be set to the number of child tasks.

After spawning, p can wait for all child tasks to finish by calling

```
void task::wait_for_all ();
```

For this function, an additional reference is needed in p, because `wait_for_all()` will execute tasks in the work set, until the reference counter of p is 1. Afterwards, it decreases the reference counter of p to 0 and returns.

The reference counter of p is decremented by using the parent variable in child tasks after `execute()` has finished. If it becomes 0, the parent is also put in the work set by TBB.

Synchronisation

Spawning the last and waiting for all child tasks can be combined by

```
void task::spawn_and_wait_for_all ( task & t );
```

Remark

The task spawned by `spawn_and_wait_for_all()` is guaranteed to be executed by the current thread.

For root tasks a special spawn routine is provided:

```
static void task::spawn_root_and_wait ( task & t );
```

Destruction

Tasks in the work set are usually automatically destroyed upon fininshing execution. The task memory is afterwards recycled for future allocations.

If this is not the case, e.g. due to special handling of the reference counter, tasks should be explicitly destroyed using

```
static void task::destroy ( task & t );
```

The reference counter of t must be 0. If a parent of t exists, its reference counter is decreased.

Example

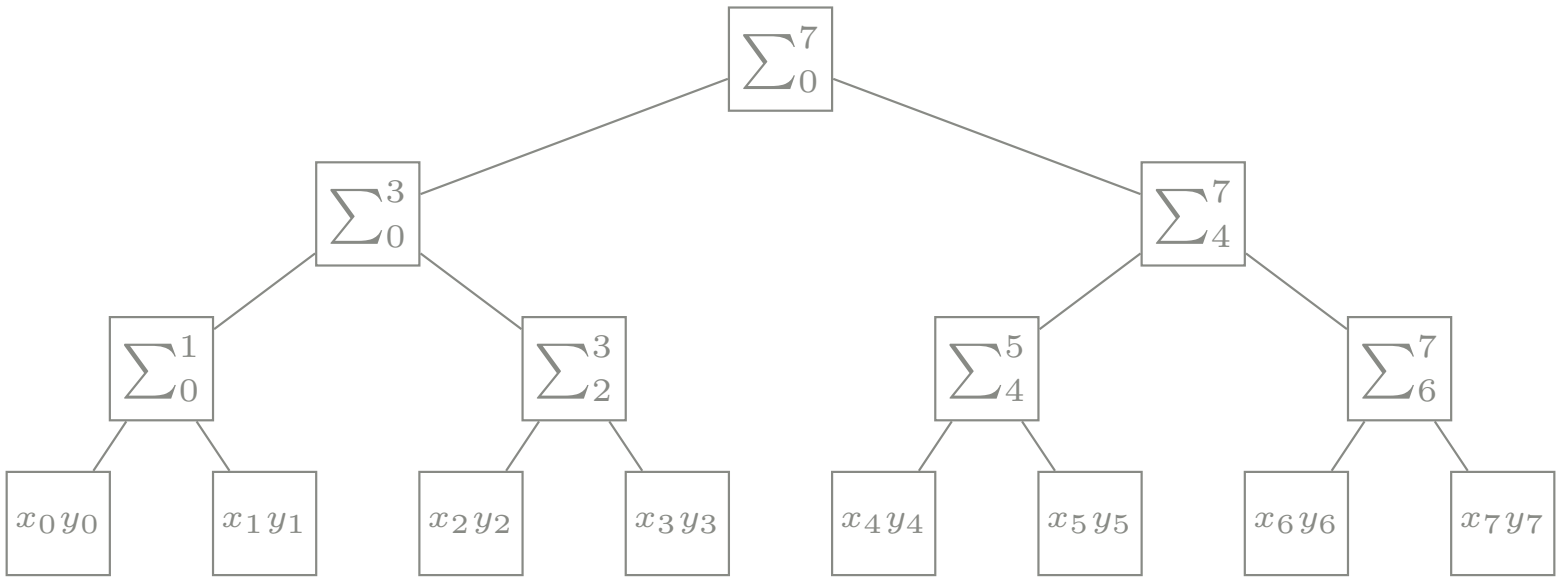
Compute $\langle x, y \rangle$ recursively, while each recursive call forms a new task:

```
class dot_task_t : public task {
public:
  const std::vector< double > & x, & y;
  const size_t                  lb, ub;    // interval [lb,ub)
  double &                      res;       // holds locally computed result

  task * execute () {
    if ( ub - lb < MIN_SIZE ) {
      res = std::inner_product( & x[lb], & x[ub], & y[lb], 0.0 );
    } else {
      double       res1   = 0;
      double       res2   = 0;
      dot_task_t & child1 = * new ( allocate_child() ) dot_task_t( x, y, lb, (ub+lb)/2, res1 );
      dot_task_t & child2 = * new ( allocate_child() ) dot_task_t( x, y, (ub+lb)/2, ub, res2 );

      set_ref_count( 3 );
      spawn( child2 );
      spawn_and_wait_for_all( child1 );

      res = res1 + res2;
      return nullptr;
} } };

double parallel_dot ( const std::vector< double > & x, const std::vector< double > & y ) {
  double       res  = 0;
  dot_task_t & root = * new ( task::allocate_root() ) dot_task_t( x, y, 0, x.size(), res );

  task::spawn_root_and_wait( root );
  return res;
}
```

On an Intel XeonPhi 5110P, the speedup of `parallel_dot()` for $n = 10^8$ is 12.63x compared to 2.56x for `parallel_reduce()`.

Scheduling

The task graph for the dot-product example with $n = 8$ and `MIN_SIZE` $= 1$ is:



The computation starts with the $\sum_0^7$ task.

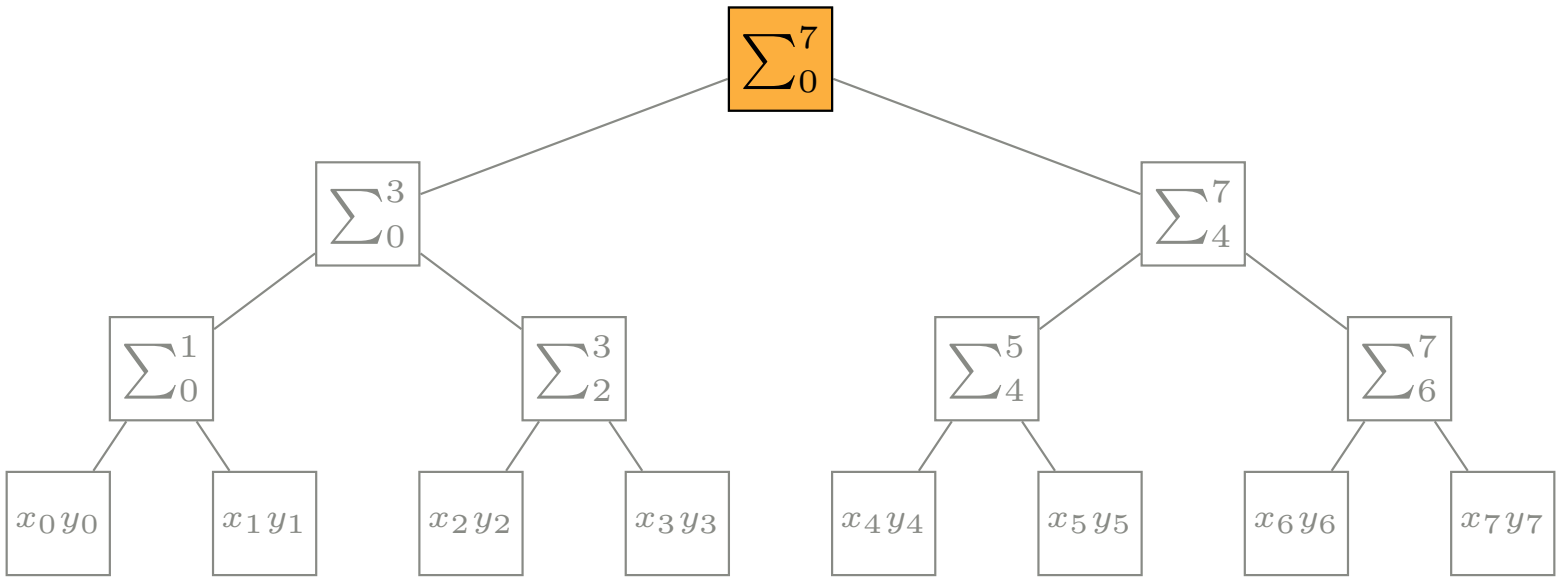
The task scheduler of TBB uses a mixed DFS/BFS scheduling strategy to optimise cache locality and parallel execution:

- tasks per thread are executed via DFS (preserves cache locality),
- task stealing follows a BFS strategy (increases parallel degree)

Some methods are provided by TBB to influence or optimise task scheduling.

Scheduling

The task graph for the dot-product example with $n = 8$ and `MIN_SIZE` $= 1$ is:



The computation starts with the $\sum_0^7$ task.

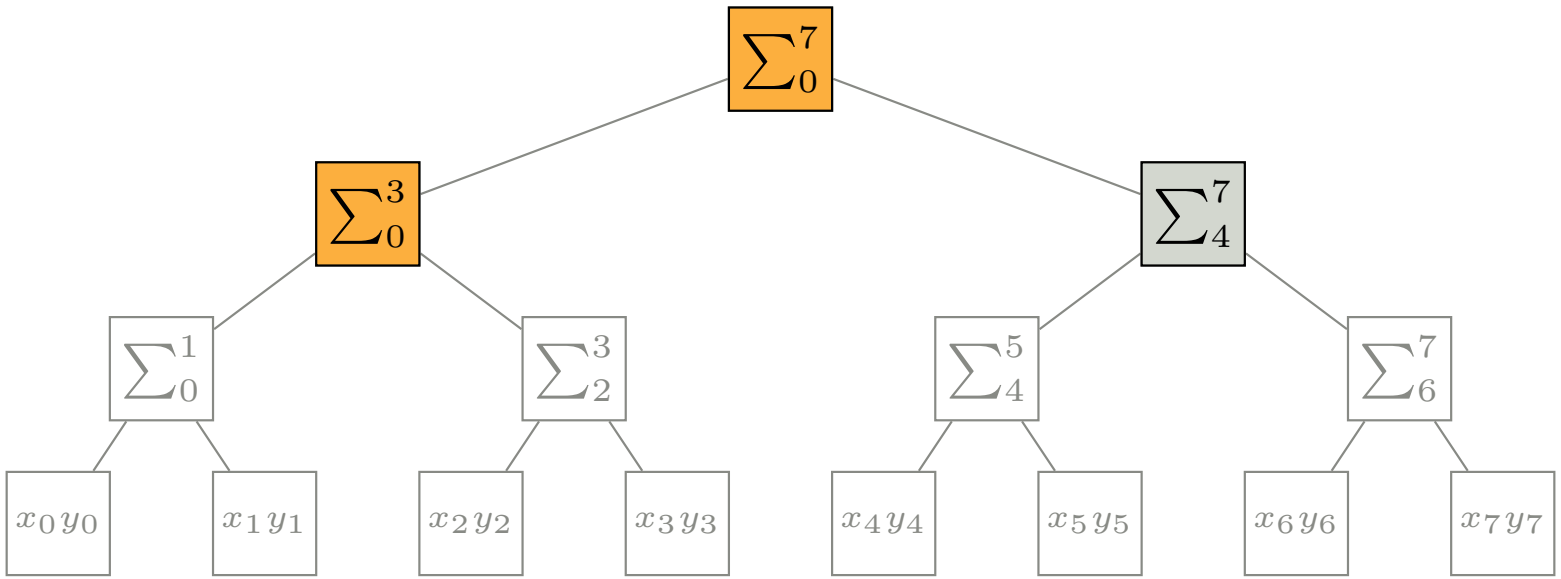
The task scheduler of TBB uses a mixed DFS/BFS scheduling strategy to optimise cache locality and parallel execution:

- tasks per thread are executed via DFS (preserves cache locality),
- task stealing follows a BFS strategy (increases parallel degree)

Some methods are provided by TBB to influence or optimise task scheduling.

Scheduling

The task graph for the dot-product example with $n = 8$ and `MIN_SIZE` $= 1$ is:



The computation starts with the $\sum_0^7$ task.

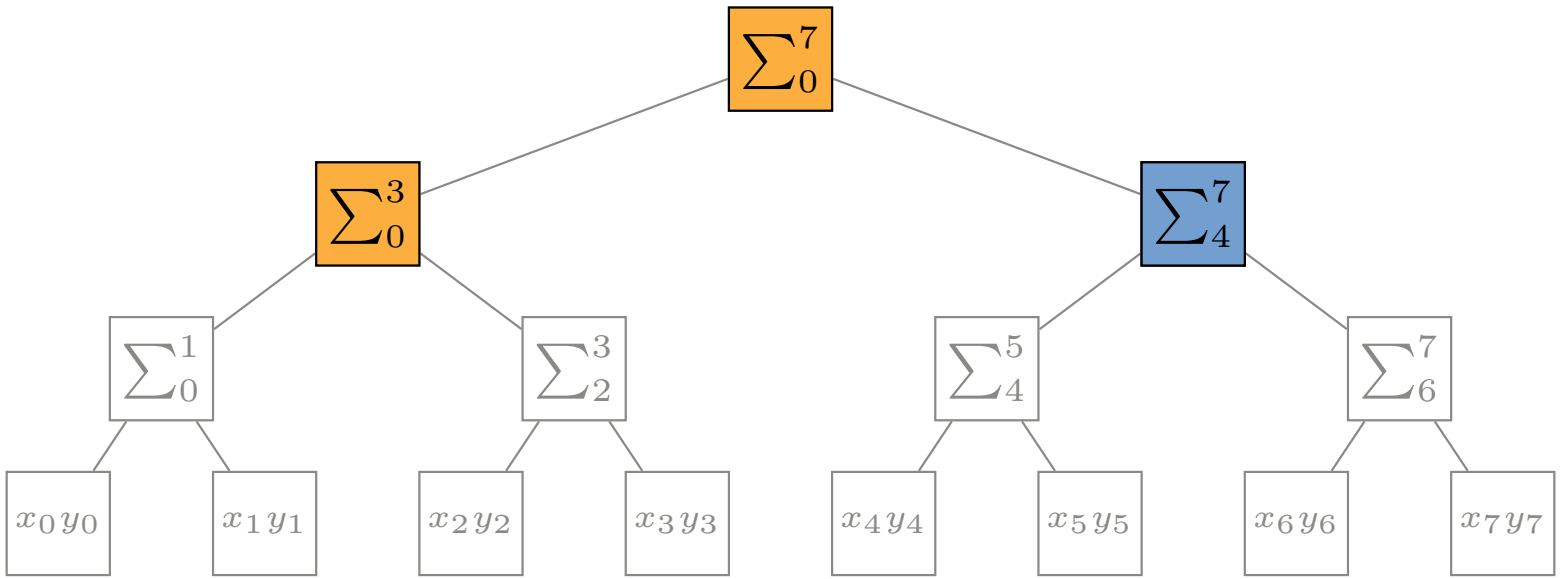
The task scheduler of TBB uses a mixed DFS/BFS scheduling strategy to optimise cache locality and parallel execution:

- tasks per thread are executed via DFS (preserves cache locality),
- task stealing follows a BFS strategy (increases parallel degree)

Some methods are provided by TBB to influence or optimise task scheduling.

Scheduling

The task graph for the dot-product example with $n = 8$ and `MIN_SIZE` $= 1$ is:



The computation starts with the $\sum_0^7$ task.

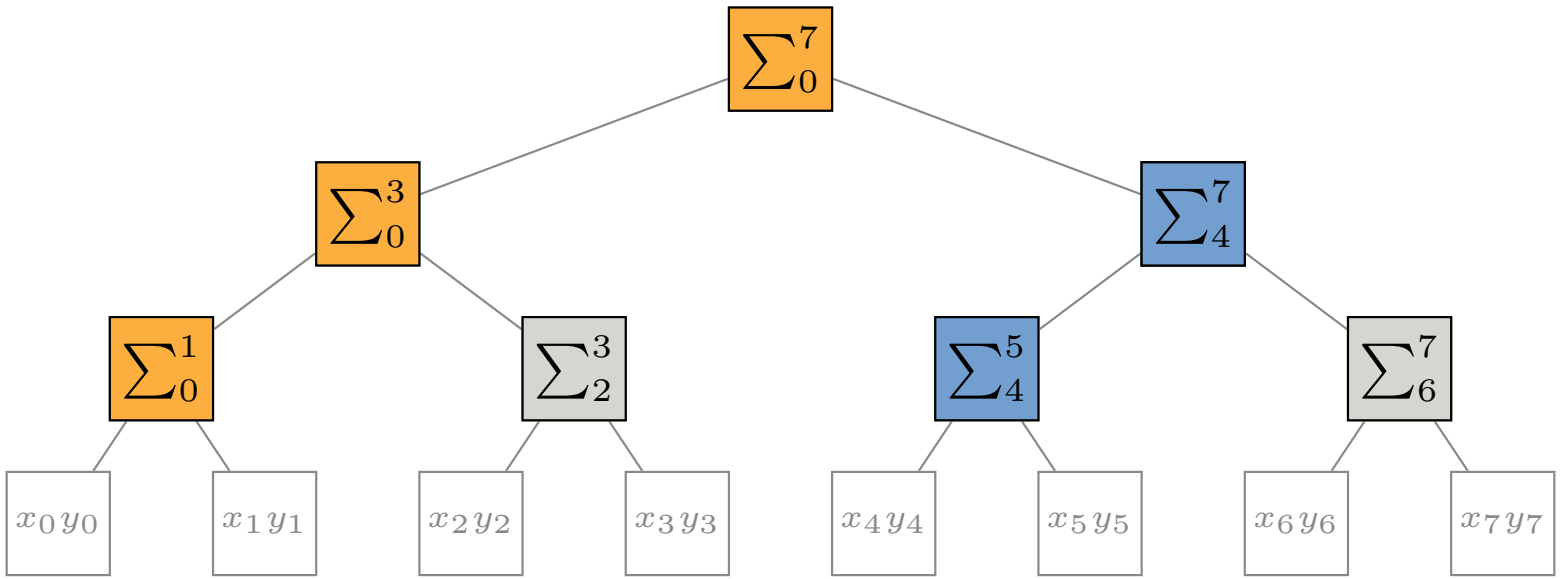
The task scheduler of TBB uses a mixed DFS/BFS scheduling strategy to optimise cache locality and parallel execution:

- tasks per thread are executed via DFS (preserves cache locality),
- task stealing follows a BFS strategy (increases parallel degree)

Some methods are provided by TBB to influence or optimise task scheduling.

Scheduling

The task graph for the dot-product example with $n = 8$ and `MIN_SIZE` $= 1$ is:



The computation starts with the $\sum_0^7$ task.

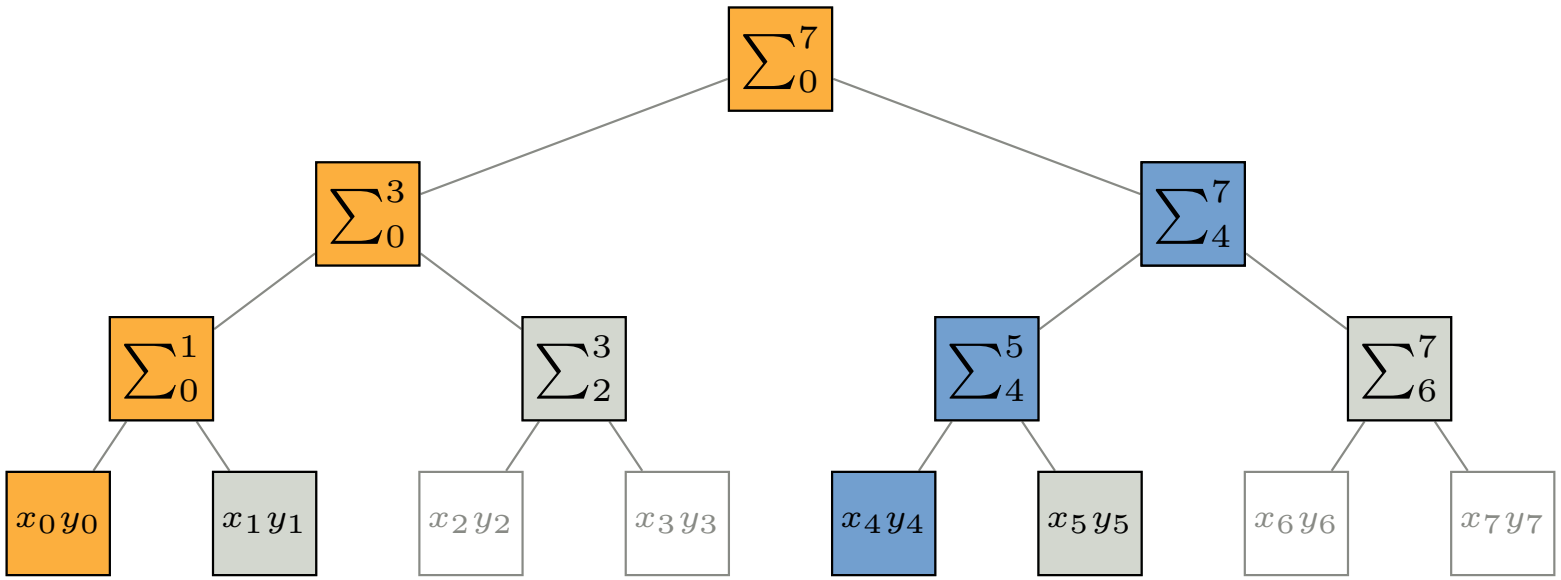
The task scheduler of TBB uses a mixed DFS/BFS scheduling strategy to optimise cache locality and parallel execution:

- tasks per thread are executed via DFS (preserves cache locality),
- task stealing follows a BFS strategy (increases parallel degree)

Some methods are provided by TBB to influence or optimise task scheduling.

Scheduling

The task graph for the dot-product example with $n = 8$ and `MIN_SIZE` $= 1$ is:



The computation starts with the $\sum_0^7$ task.

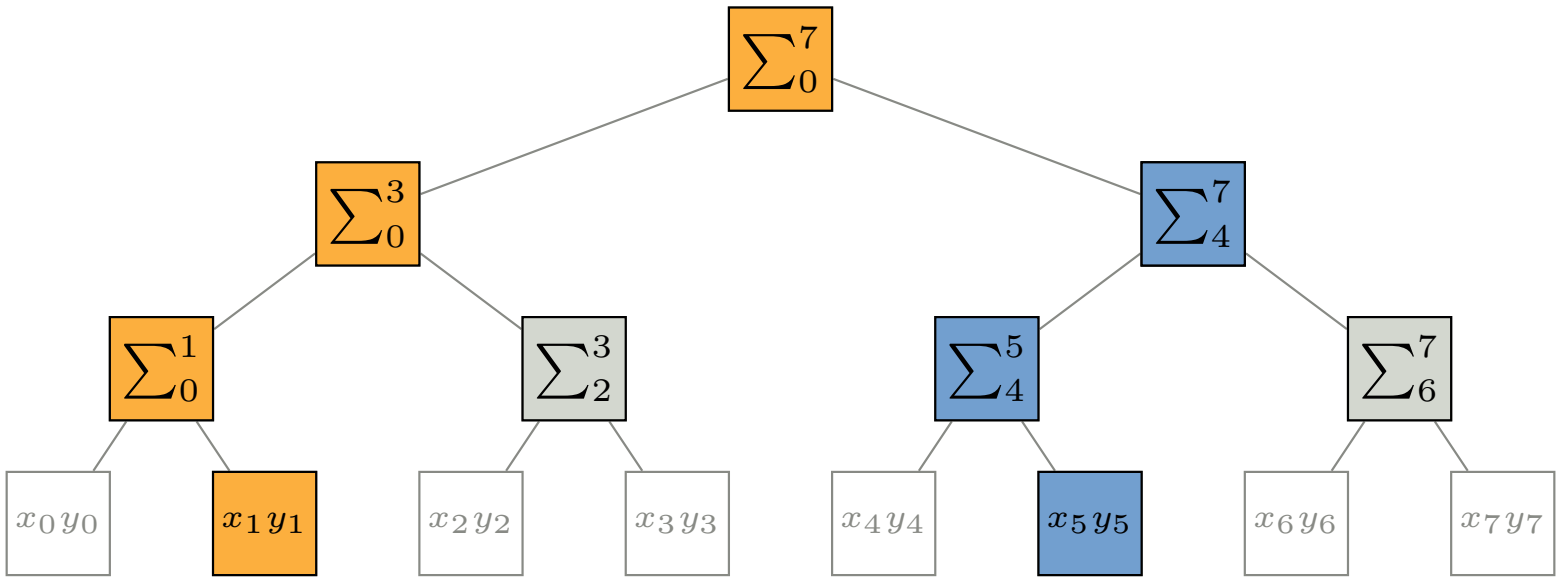
The task scheduler of TBB uses a mixed DFS/BFS scheduling strategy to optimise cache locality and parallel execution:

- tasks per thread are executed via DFS (preserves cache locality),
- task stealing follows a BFS strategy (increases parallel degree)

Some methods are provided by TBB to influence or optimise task scheduling.

Scheduling

The task graph for the dot-product example with $n = 8$ and `MIN_SIZE` $= 1$ is:



The computation starts with the $\sum_0^7$ task.

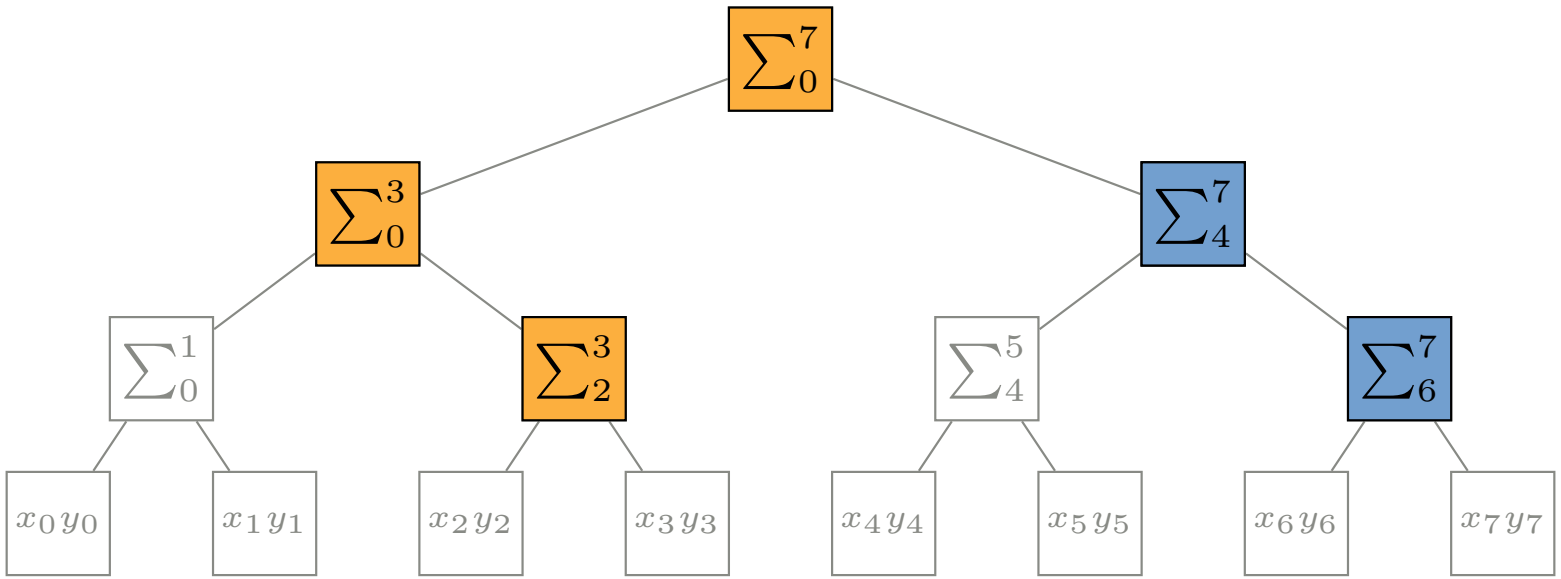
The task scheduler of TBB uses a mixed DFS/BFS scheduling strategy to optimise cache locality and parallel execution:

- tasks per thread are executed via DFS (preserves cache locality),
- task stealing follows a BFS strategy (increases parallel degree)

Some methods are provided by TBB to influence or optimise task scheduling.

Scheduling

The task graph for the dot-product example with $n = 8$ and `MIN_SIZE` $= 1$ is:



The computation starts with the $\sum_0^7$ task.

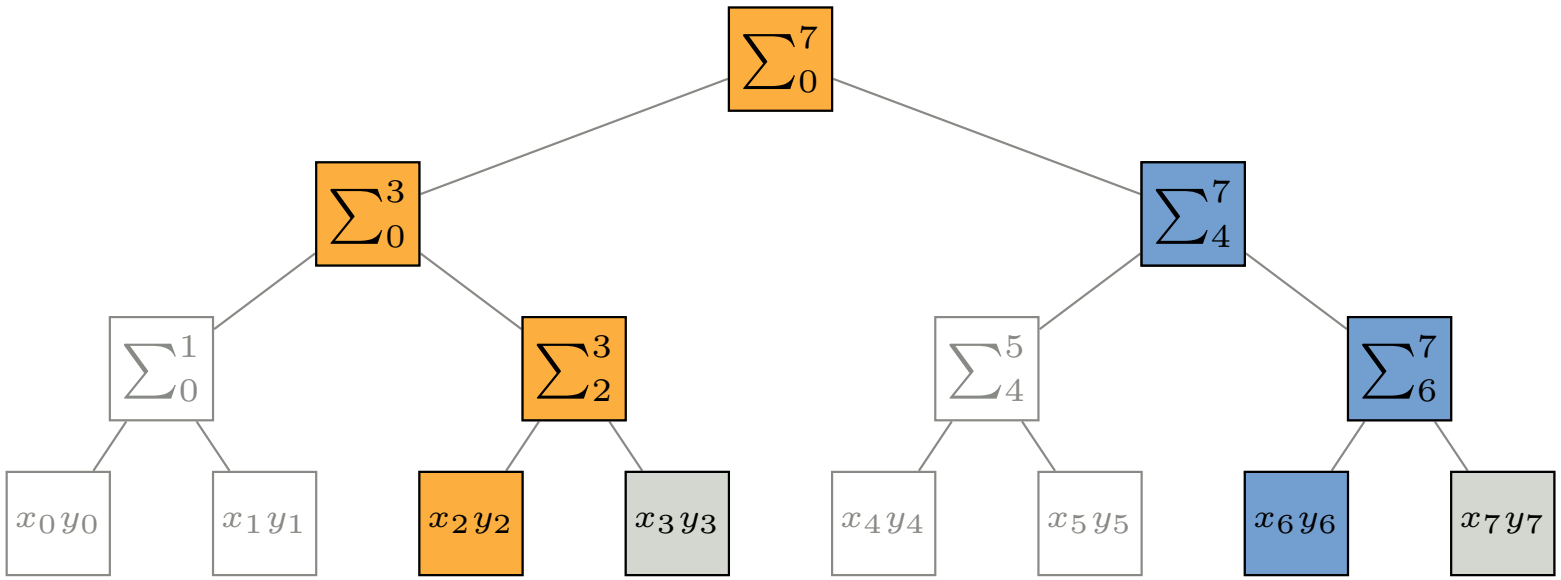
The task scheduler of TBB uses a mixed DFS/BFS scheduling strategy to optimise cache locality and parallel execution:

- tasks per thread are executed via DFS (preserves cache locality),
- task stealing follows a BFS strategy (increases parallel degree)

Some methods are provided by TBB to influence or optimise task scheduling.

Scheduling

The task graph for the dot-product example with $n = 8$ and `MIN_SIZE` $= 1$ is:



The computation starts with the $\sum_0^7$ task.

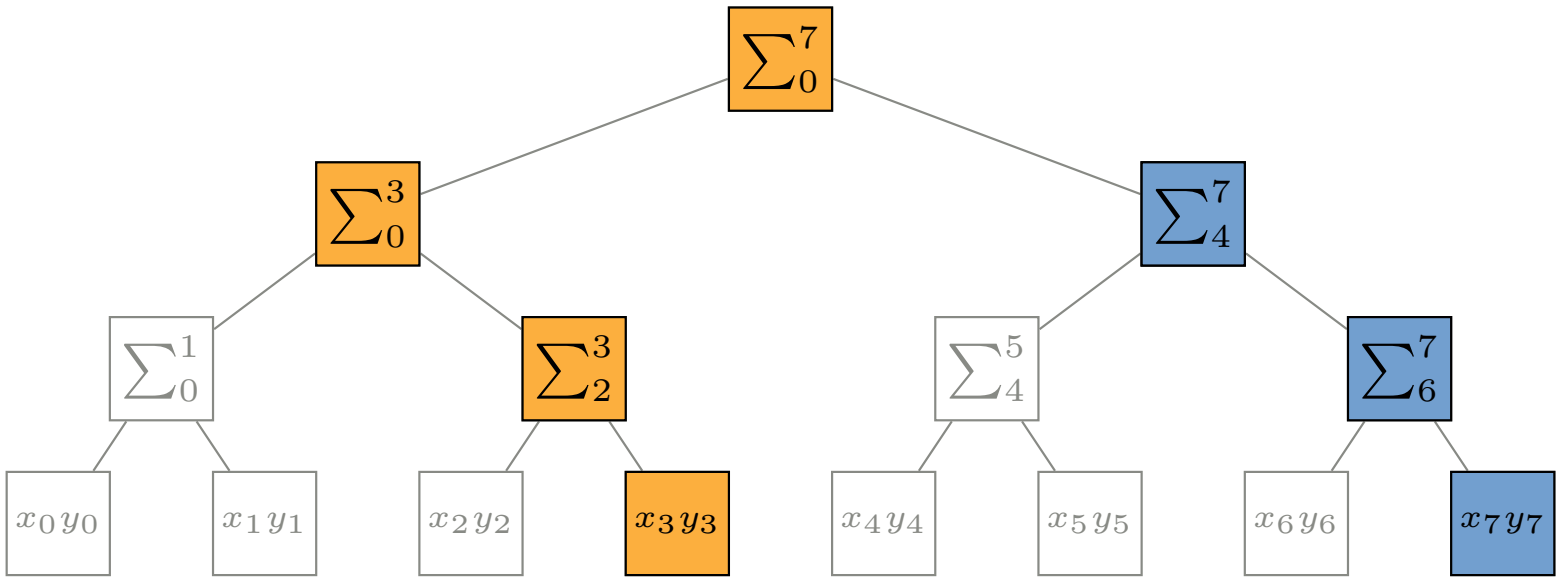
The task scheduler of TBB uses a mixed DFS/BFS scheduling strategy to optimise cache locality and parallel execution:

- tasks per thread are executed via DFS (preserves cache locality),
- task stealing follows a BFS strategy (increases parallel degree)

Some methods are provided by TBB to influence or optimise task scheduling.

Scheduling

The task graph for the dot-product example with $n = 8$ and `MIN_SIZE` $= 1$ is:



The computation starts with the $\sum_0^7$ task.

The task scheduler of TBB uses a mixed DFS/BFS scheduling strategy to optimise cache locality and parallel execution:

- tasks per thread are executed via DFS (preserves cache locality),
- task stealing follows a BFS strategy (increases parallel degree)

Some methods are provided by TBB to influence or optimise task scheduling.

Continuation Passing

A thread, which executes a task calling `spawn_and_wait_for_all()`, will executes other tasks while the child tasks are running. If the parent task may finally proceed, the corresponding thread could be busy, preventing immediate execution of the parent task.

This can be overcome by using a *continuation* task, i.e. a separate task which continues execution after all child tasks of the original parent task have finished:

```
task * dot_task_t::execute () {
  if ( ub - lb < MIN_SIZE ) { ... }
  else {
    cont_t & cont   = * new ( allocate_continuation() ) cont_t( res );
    task &   child1 = * new ( cont.allocate_child() ) dot_task_t( x, y, lb, (ub+lb)/2, cont.res1 );
    task &   child2 = * new ( cont.allocate_child() ) dot_task_t( x, y, (ub+lb)/2, ub, cont.res2 );

    cont.set_ref_count( 2 );
    spawn( child2 );
    spawn( child1 );
    return nullptr;
} }
```

The child tasks will have the continuation task as their parent. As its reference counter is 2, it is scheduled immediatly after the last child finishes. If the current thread is busy, another thread may steal the continuation task and execute it.

Remark

Using a continuation task for the dot-product, the speedup increases to 23.93x.

Scheduler Bypass

Up to now, the return value of `execute()` was always `nullptr`.

It can also be a pointer to a task, which is then chosen as the next task to be executed by the current thread.

```
task * dot_task_t::execute () {
  if ( ub - lb < MIN_SIZE ) { ... }
  else {
    cont_t & cont   = * new ( allocate_continuation() ) cont_t( res );
    task &   child1 = * new ( cont.allocate_child() ) dot_task_t( x, y, lb, (ub+lb)/2, cont.res1 );
    task &   child2 = * new ( cont.allocate_child() ) dot_task_t( x, y, (ub+lb)/2, ub, cont.res2 );

    cont.set_ref_count( 2 );
    spawn( child2 );

    return & child1;
} }
```

Before, the `child1` task was put in the work set of the current thread, and taken out after `execute()` has finished.

Both operations are eliminated by handing `child1` directly to the task scheduler.

Remark

For the dot-product computation, the speedup increases to 29.03x, when bypassing the scheduler.

Recycling

Bypassing the scheduling process can be combined with bypassing task allocation and deallocation.

```
task * dot_task_t::execute () {
  if ( ub - lb < MIN_SIZE ) { ... }
  else {
    cont_t & cont   = * new ( allocate_continuation() ) cont_t( res );
    task &   child2 = * new ( cont.allocate_child() ) dot_task_t( x, y, (ub+lb)/2, ub, cont.res2 );

    recycle_as_child_of( cont );
    ub  = (ub+lb)/2;
    res = cont.res1;

    cont.set_ref_count( 2 );
    spawn( child2 );

    return this;
} }
```

The current task t is updated with the data of `child1`. Furthermore,

```
void task::recycle_as_child_of( task & new_parent );
```

will set the continuation task of t as its new parent.

Finally, the current task is reexecuted immediately by returning a pointer to it.

Remark

If the parent task is recycled as child during dot-product computation, the speedup increases to 31.70x.

Recycling

Recycling a parent task as a child task is especially efficient, if the child can reuse data from the parent.

If on the other hand, a continuation task may reuse data, a task can also be recycled as its own continuation via

```
void task::recycle_as_continuation ();
```

A potential race condition exists, if all child tasks of a recycled task t have finished, at which point t is scheduled to be executed again.

If t has not finished its own `execute()` function, it is run in parallel. As this may lead to various side effects, TBB requires that such situations must not occur.

One way to prevent this execution overlap is to *not* spawn one of the child tasks but return it from `execute()`.

Another way is to set the reference count of the continuation task to $k + 1$ where k is the number of child tasks and use

```
void task::recycle_as_safe_continuation ();
```

which avoids the race condition.

Shared Queue

Each worker thread in TBB has its local work set of tasks. If not using scheduler bypass or when the last child of a parent task finished, the thread will look first at this set for a new task to execute.

TBB also has a *shared set* for all threads, from which it will take tasks if the thread local set is empty. Only if this shared set is also empty, it will try to steal tasks from other sets.

New tasks may be explicitly put into the shared task set via

```
static void task::enqueue ( task & t );
```

In contrast to the thread local work set, which is handled in a *last-in first-out* order, the shared set is accessed (roughly) in *first-in first-out* order.

This behaviour for the shared set ensures some *fairness* for task execution, i.e. eventually the enqueued task will be executed. Tasks in the thread-local set may not be scheduled until another task explicitly waits for them.

Furthermore, even if the number of worker threads is zero, a thread is started to handle tasks in the shared set.

Affinity

Thread affinity and, assuming a processor bound thread, also cache affinity for tasks is supported by TBB through a set of `task` functions.

The affinity id of an executed task is automatically send to the task via

```
virtual void task::note_affinity ( affinity_id  id );
```

By overloading this function in a user defined task, the affinity id can be recorded and set for a later spawned task using

```
void task::set_affinity ( affinity_id  id );
```

The task scheduler will afterwards use the thread with the same id to execute the task. For this, the affinity id has to be set *before* the task is spawned:

```
task & child = * new( allocate_child() ) child_task_t( ... );

child.set_affinity( my_id );
spawn( child );
```

Access to the current affinity id of a task is provided with

```
affinity_id task::affinity () const;
```

Remark

Variables of type `affinity_id` must default to 0 or hold values from `note_affinity()`.

Priority

For tasks in the shared queue a priority may be set as an optional argument to `enqueue`:

```
static void task::enqueue ( task &     t,
                            priority_t p );
```

TBB supports three different levels of priority:

- `priority_high`,
- `priority_normal`,
- `priority_low`.

Tasks of a higher priority will be first taken out of the shared queue and scheduled for execution.

The priority of the task is fixed when calling `enqueue()`, i.e. can not be changed afterwards.

Remark

Spawned tasks will have the priority of their task group (see below).

Task Groups

All tasks are part of a task group, represented by objects of type `task_group_context`.

Each creation of an `task_scheduler_init`, e.g. during explicit TBB initialisation, also creates an implicit task group object. All tasks spawned during the lifetime of that object, automatically belong to this task group:

```
{
  task_scheduler_init  tsi1( 4 );     // task group 1
  spawn_root_and_wait( ... );

  {
    task_scheduler_init  tsi2( 8 );   // task group 2
    spawn_root_and_wait( ... );
  }
}

{
  task_scheduler_init  tsi3( 2 );     // task group 3
  spawn_root_and_wait( ... );
}
```

All task groups in a process form a forest of trees, i.e. multiple root task groups may exist, each having various sub groups.

The hierarchy defined by the tree also determines certain properties of the task groups, e.g. for cancellation or priority.

Task Groups

A root task may also be explicitly assigned to a task group by providing the `task_group_context` object during task allocation:

```
task_group_context  g;
task &              t = * new( task::allocate_root( g ) ) my_task_t( ... );
```

All child tasks allocated with `task::alocate_child()` will automatically belong to the same task group as the parent task.

Changing the task group of a task can be done using

```
void task::change_group ( task_group_context & g );
```

The task group of a task is accessed via

```
task_group_context * task::group ();
```

Cancellation

All tasks in a task group and of all sub groups can be cancelled using

```
bool task::cancel_group_execution ();
```

Tasks already running are not directly affected by this. However, they may query the current cancellation status via

```
bool task::is_cancelled ();
```

and stop further execution, e.g.:

```
task * execute () {
  while ( ! is_cancelled() ) {
    proceed_computation();
  }
}
```

All other tasks, i.e. tasks still waiting for execution, will not call `execute()` after a cancellation request was issued.

If the task group already received a cancellation request, the return value of `cancel_group_execution()` is `false`.

Remark

Throwing an exception within a task will also cancel further execution within the corresponding task group.

Priority

Associated with a task group is a *priority* level of the same type and values as for tasks in the shared queue, i.e. either `priority_high`, `priority_normal` or `priority_low`.

The priority of a task group affects all tasks in the group and in all sub groups. Tasks in a task group of a higher priority will be executed before tasks in group with a lower priority.

To set the priority of a task group either use

```
void task_group_context::set_priority ( priority_t p );
```

or

```
void task::set_group_priority ( priority_t p );
```

Similarly, the priority of a task group may be accessed using one of

```
priority_t task_group_context::priority ();
priority_t task::group_priority ();
```

In contrast to the priority of an `enqueue`'d task, the priority of task groups may be changed at any time. However the effect may not be immediate, e.g. tasks with a lower priority may still be scheduled first.

Task Lists

Spawning several tasks can also be combined using the TBB type *task_list*:

```
static void spawn                  ( task_list & list );
       void spawn_and_wait_for_all ( task_list & list );
static void spawn_root_and_wait    ( task_list & list );
```

The class `task_list` supports basic STL container functions:

```
class task_list {
public:
  bool    empty     () const;         // return true if empty
  void    push_back ( task & task ); // insert task at end of list
  task &  pop_front ();               // remove and return first task in list
  void    clear     ();               // remove all tasks from list
};
```

Spawning task lists may be more efficient than spawning each task separatly:

```
task_list list;

set_ref_count( nchildren + 1 );
for ( int i = 0; i < nchildren; ++i )
  list.push_back( * new( allocate_child() ) child_task_t( ... );

spawn_and_wait_for_all( list );
```

Remark

Recursive task spawning should be prefered over long task lists.

DAG Computations

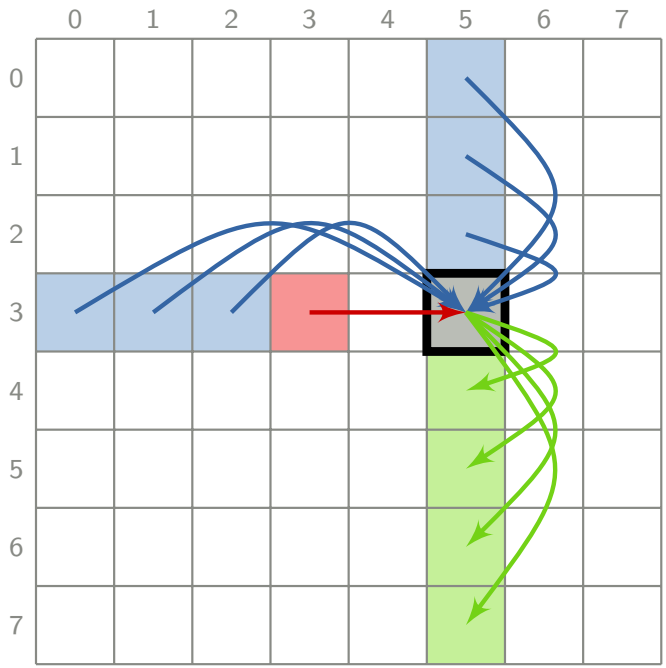
Up to now, task creation was coupled with task spawning due to recursive parallelism, which also reflected in the relation between parent and child tasks.

The task concept can also be viewed in terms of *successors* and *predeccesors*, where a successor task is spawned after a predeccesor task has finished.

This relation can be expressed in terms of a graph, where nodes correspond to tasks and directed edges to dependencies between them. Since no cyclic dependencies may exists, such tasks are *directed acyclic graphs* or *DAGs*.

For the block-wise LU factorisation of a dense matrix $A \in \mathbb{R}^{n \times n}$ the incoming and outgoing dependencies for a single matrix block are shown to the right.

Incoming dependencies are due to needed matrix updates (blue) and the diagonal factorisation (red). Outgoing dependencies are because of matrix updates to blocks below (green).



DAG Computations

Since several incoming dependencies exist, pre-allocation of tasks is of advantage. Here, the reference counter may be set according to the degree of incoming edges in the DAG. After each predecessor has finished, it can decrement the reference counter of all successors. If this reaches zero, the task is scheduled for execution.

For the block-wise LU factorisation, matrix block A_{ij} has $2 \cdot \min(i, j)$ dependencies due to matrix updates. In addition, off-diagonal blocks depend on the factorisation of the diagonal block.

This leads to the following implementation, where for each matrix block a task is pre-allocated, either for factorisation or matrix solves, and initialised with the corresponding number of dependencies:

```
std::vector< std::vector< task * > >  tasks;

for ( size_t i = 0; i < b; ++i ) {
  // factorise diagonal block
  tasks[i][i] = new( task::allocate_root() ) fac_task_t( A, i, tasks, mutexes );
  tasks[i][i]->set_ref_count( 2*i );

  // solve offdiagonal blocks
  for ( size_t j = i+1; j < b; ++j ) {
    tasks[i][j] = new( task::allocate_root() ) solve_lower_task_t( A, i, j, tasks, mutexes );
    tasks[i][j]->set_ref_count( 1 + 2*i );

    tasks[j][i] = new( task::allocate_root() ) solve_upper_task_t( A, j, i, tasks, mutexes );
    tasks[j][i]->set_ref_count( 1 + 2*i );
} }
```

DAG Computations

The implementation of a factorisation task consists of the computation of all updates, the actual point-wise factorisation and spawning successor tasks:

```
class fac_task_t : public task {
private:
  ...
  task_matrix_t &   tasks;

public:
  ...
  task * execute () {
    const size_t  b = A.n/BLOCK_SIZE;
    Matrix        A_ii( BLOCK_SIZE );

    // spawn update tasks
    spawn_updates( A, this, i, i, i );

    // factorise
    load_block(  A, A_ii, i, i );
    lu( A_ii );
    store_block( A, A_ii, i, i );

    // spawn tasks below and to the right
    for ( size_t  k = i+1; k < b; ++k ) {
      if ( tasks[i][k]->decrement_ref_count() == 0 ) spawn( * tasks[i][k] );
      if ( tasks[k][i]->decrement_ref_count() == 0 ) spawn( * tasks[k][i] );
    }

    return nullptr;
} };
```

Tasks for matrix solves are similar, e.g. first apply updates, then perform matrix solve and finally spawn successors.

DAG Computations

The function *spawn_updates()* spawns tasks for all matrix updates

$$A_{ij} = A_{ij} - \sum_{k=0}^{\ell-1} L_{ik} U_{kj}$$

with $\ell = \min(i, j)$.

In contrast to factorisation and solve tasks, update tasks are created and spawned when needed:

```
void spawn_updates ( Matrix & A, task * t
                     size_t i, size_t j, size_t l ) {
  t->set_ref_count( l+1 );

  // spawn updates for A_ij = A_ij - Σ_{0≤k<ℓ} L_ik U_kj
  for ( size_t  k = 0; k < l; ++k )
    t->spawn( * new( t->allocate_child() )
              update_task_t( A, i, j, k ) );

  t->wait_for_all();
}
```

```
class update_task_t : public task {
...
task * execute () {
  Matrix  U_sub( BLOCK_SIZE );
  Matrix  L_sub( BLOCK_SIZE );
  Matrix  A_sub( BLOCK_SIZE );

  load_block( A, L_sub, i, k );
  load_block( A, U_sub, k, j );
  load_block( A, A_sub, i, j );

  multiply_sub( L_sub, U_sub, A_sub );

  return nullptr;
} };
```

DAG Computations

The function *spawn_updates()* spawns tasks for all matrix updates

$$A_{ij} = A_{ij} - \sum_{k=0}^{\ell-1} L_{ik} U_{kj}$$

with $\ell = \min(i, j)$.

In contrast to factorisation and solve tasks, update tasks are created and spawned when needed:

```
void spawn_updates ( Matrix & A, task * t
                     size_t i, size_t j, size_t l ) {
  t->set_ref_count( l+1 );

  // spawn updates for A_ij = A_ij - Σ_{0≤k<ℓ} L_ik U_kj
  for ( size_t  k = 0; k < l; ++k )
    t->spawn( * new( t->allocate_child() )
              update_task_t( A, i, j, k ) );

  t->wait_for_all();
}
```

```
class update_task_t : public task {
...
task * execute () {
  Matrix  U_sub( BLOCK_SIZE );
  Matrix  L_sub( BLOCK_SIZE );
  Matrix  A_sub( BLOCK_SIZE );

  load_block( A, L_sub, i, k );
  load_block( A, U_sub, k, j );
  load_block( A, A_sub, i, j );

  multiply_sub( L_sub, U_sub, A_sub );

  return nullptr;
} };
```

Remark

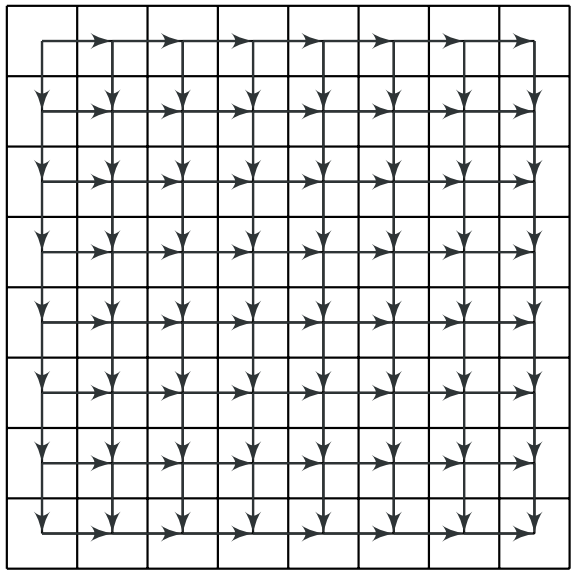
Updating a matrix block represents a critical section (see below).

DAG Computations

The node to start the whole computation corresponds to the matrix block A_{00}. The end of the process is formed by the task for matrix block $A_{b-1,b-1}$.

Furthermore, by construction, a path exists from the start to the final node in the DAG.

The execution of the DAG is initiated by the final node, with the start node as the first task to execute.



As before, an additional reference is needed for `spawn_and_wait_for_all()`. Since this prevents automatic execution of the final node, it is performed manually, as is the destruction of the task:

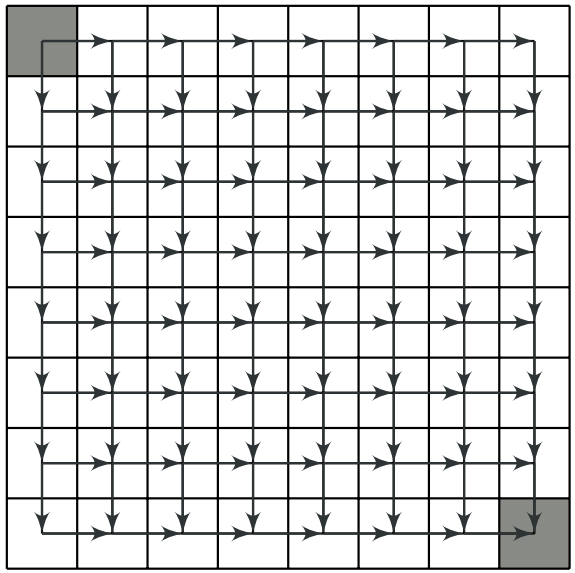
```
tasks[b-1][b-1]->increment_ref_count();
tasks[b-1][b-1]->spawn_and_wait_for_all( * tasks[0][0] );
tasks[b-1][b-1]->execute();
task::destroy( * tasks[b-1][b-1] );
```

DAG Computations

The node to start the whole computation corresponds to the matrix block A_{00}. The end of the process is formed by the task for matrix block $A_{b-1,b-1}$.

Furthermore, by construction, a path exists from the start to the final node in the DAG.

The execution of the DAG is initiated by the final node, with the start node as the first task to execute.



As before, an additional reference is needed for `spawn_and_wait_for_all()`. Since this prevents automatic execution of the final node, it is performed manually, as is the destruction of the task:

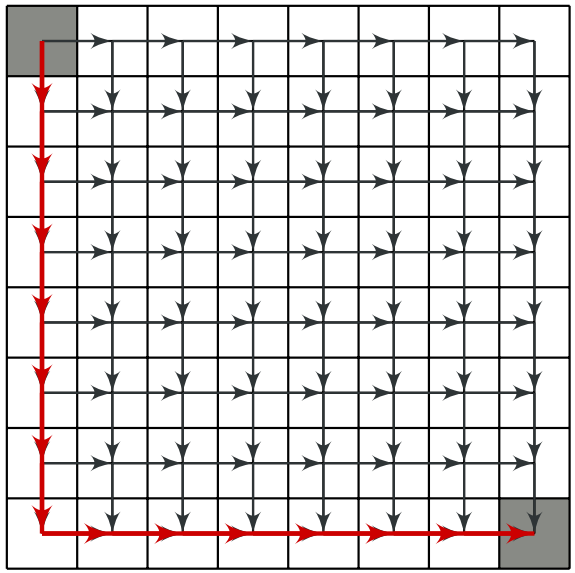
```
tasks[b-1][b-1]->increment_ref_count();
tasks[b-1][b-1]->spawn_and_wait_for_all( * tasks[0][0] );
tasks[b-1][b-1]->execute();
task::destroy( * tasks[b-1][b-1] );
```

DAG Computations

The node to start the whole computation corresponds to the matrix block A_{00}. The end of the process is formed by the task for matrix block $A_{b-1,b-1}$.

Furthermore, by construction, a path exists from the start to the final node in the DAG.

The execution of the DAG is initiated by the final node, with the start node as the first task to execute.



As before, an additional reference is needed for `spawn_and_wait_for_all()`. Since this prevents automatic execution of the final node, it is performed manually, as is the destruction of the task:

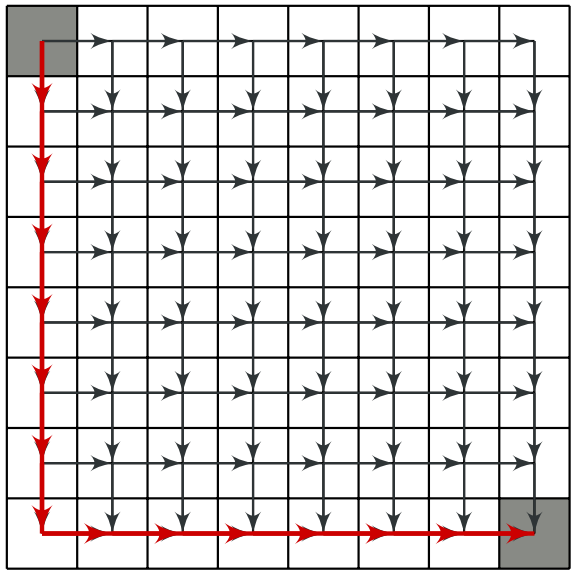
```
tasks[b-1][b-1]->increment_ref_count();
tasks[b-1][b-1]->spawn_and_wait_for_all( * tasks[0][0] );
tasks[b-1][b-1]->execute();
task::destroy( * tasks[b-1][b-1] );
```

DAG Computations

The node to start the whole computation corresponds to the matrix block A_{00}. The end of the process is formed by the task for matrix block $A_{b-1,b-1}$.

Furthermore, by construction, a path exists from the start to the final node in the DAG.

The execution of the DAG is initiated by the final node, with the start node as the first task to execute.



As before, an additional reference is needed for `spawn_and_wait_for_all()`. Since this prevents automatic execution of the final node, it is performed manually, as is the destruction of the task:

```
tasks[b-1][b-1]->increment_ref_count();
tasks[b-1][b-1]->spawn_and_wait_for_all( * tasks[0][0] );
tasks[b-1][b-1]->execute();
task::destroy( * tasks[b-1][b-1] );
```

The parallel speedup for $n = 8192$ and a block size of 64 is:

	Xeon X5650 (24 threads)	Xeon E5-2640 (24 threads)	XeonPhi 5110P (240 threads)
DAG-based:	9.11x	9.94x	72.36x

Mutual Exclusion

Mutual Exclusion

TBB implements various forms of mutexes. The differences between these mutex classes are based on certain properties mutexes may have:

Scalable: Using a *scalable* mutex will not result in a worse performance than sequential execution, e.g. due to high processor or memory bandwidth usage.

Fair: A *fair* mutex guarantees, that all threads waiting for the mutex, will eventually succeed in locking it.

Recursive: A *recursive* mutex permits a thread, who already holds a lock on a mutex to lock it again.

Yield or Block: A *yielding* mutex will only temporarily yield a processor to other threads, while a *blocking* mutex will yield the processor until the lock is aquired.

Remark

Unscalable mutexes may be faster if the lock contention is low. Similarly, fairness requires more overhead than unfair behaviour. Reactivating a blocking thread also taes some time. Hence, yielding shoud be preferred if the waiting time for a mutex is short.

Mutual Exclusion

In the following table, the different mutex types of TBB with their properties are shown:

	Scalable	Fair	Recursive	Long Wait
`mutex`	OS dep.	OS dep.	no	blocks
`recursive_mutex`	OS dep.	OS dep.	yes	blocks
`spin_mutex`	no	no	no	yields
`queuing_mutex`	yes	yes	no	yields
`null_mutex`	/	yes	yes	never

Both, `mutex` and `recursive_mutex` are based on the implementation of mutexes of the operation system.

All other mutexes are implemented in TBB, i.e. no operating system functionality is used.

A special case is `null_mutex`, which can be used as a dummy.

Scoped Locking

Locking and unlocking mutexes manually easily results in program errors, e.g. due to forgotton mutex release. Also, unlocking mutexes in case of an exception is tedious and error-prone.

Instead, mutexes should only accessed via `scoped locks`.

Each mutex type M in TBB has an associated type for a scoped lock:

```
M::scoped_lock
```

with the following interface:

```
class M::scoped_lock {
  scoped_lock  ();              // default constructor without mutex
  scoped_lock  ( M & m );       // constructor: lock mutex m
  ~scoped_lock ();              // unlock mutex if previously locked

  void acquire     ( M & m );  // lock mutex m
  bool try_acquire ( M & m );  // try to lock mutex m; return true if successful
  void release     ();         // unlock previously locked mutex
};
```

By definition, unlocking is coupled with the destruction of the scoped lock, and hence bound to the object lifetime.

Therefore, scoped locks should be *auto variables*, e.g. not allocated via `new`.

Scoped Locking

In a typical scenario, the scoped lock is constructed at the beginning of a critical section and automatically destroyed at its end:

```
spin_mutex  m;
...
{                                       // critical section
  spin_mutex::scoped_lock lock( m );    // acquire lock
  ...
}                                       // release lock automatically
...
```

or

```
void f ( mydata_t & d, recursive_mutex & m ) {
  recursive_mutex::scoped_lock lock( m );  // acquire lock
  ...
}                                          // release lock automatically
```

C++11 Interface

TBB mutexes also support the standard C++11 mutex functions:

```
void M::lock     ();  // lock mutex
bool M::try_lock ();  // try to lock mutex
void M::unlock   ();  // unlock mutex
```

This also allows to use C++11 `std::lock_guard` together with TBB mutexes.

Reader/Writer Locks

TBB extends mutexes to distinguish between *readers* and *writers* with respect to a critical section. Multiple readers may hold a lock, but only a single writer.

The extension affects only the special mutex classes from TBB, not the OS related mutexes.

For each TBB mutex, a corresponding Reader/Writer (RW) lock exists:

	Scalable	Fair	Recursive	Long Wait
`spin_rw_mutex`	no	no	no	yields
`queuing_rw_mutex`	yes	yes	no	yields
`null_rw_mutex`	/	yes	yes	never

The scoped lock interface is also extended, now accepting a boolean parameter *writer* with each mutex, indicating whether the lock shall be a write lock (*true*) or a reader lock (*false*):

```
class M::scoped_lock {
  scoped_lock      ( M & m, bool writer );
  void acquire     ( M & m, bool writer );
  bool try_acquire ( M & m, bool writer );
};
```

Reader/Writer Locks

Reader locks may also be upgraded to writer locks and vice versa:

```
bool M::scoped_lock::upgrade_to_writer   ();  // upgrade to write lock
bool M::scoped_lock::downgrade_to_reader ();  // downgrade to reader lock
```

In both cases, either a reader (for `upgrade_to_writer()`) or a writer lock (for `downgrade_to_reader()`) must already be acquired. Otherwise, the effect is undefined.

When upgrading a reader lock to a writer lock, it will block until the last reader releases the lock.

Remark

Both functions return `false` if the lock was released and reacquired, e.g. via `release()`/`acquire()`, and `true` otherwise.

An example usage of RW locks are read-only and write sections in a program:

```
{
  M::scoped_lock  lock( m, false );  // acquire reader lock
  ...                                // access data read-only
  {
    lock.upgrade_to_writer();
    ...                              // write data
    lock.downgrade_to_reader();
  }
  ...                                // access read-only again
}
```

Containers

Containers

Standard C++ or C++11 STL containers are *not* multi-thread safe if modified. Hence, access has to be guarded by a mutex, preventing efficient parallel operations.

TBB on the other hand, implements several containers, which permit concurrent access, e.g. insertion, deletion, by using either

- a *fine-grained* locking mechanism, i.e. locking only small parts of the data structure which are actually changed or
- *lock-free* mechanisms, where threads notice changes by other threads and handle the effects of these changes automatically.

The implemented containers fall into the following categories:

dynamic arrays:	implementes dynamic resize of an array
associative arrays:	a *key* type is used to access data,
queues:	implements *first-in first-out* data access.

Remark

Concurrent access comes with a cost, i.e. worse sequential performance. Therefore, concurrent containers should be used, if the advantage of concurrency and the corresponding speedup outwheighs the additional costs.

Dynamic Arrays

The concurrent version of `std::vector` in TBB is *`concurrent_vector`*.

It supports concurrent growing of the vector, i.e. several threads increase the size simultaneously.

For this, the following methods are provided:

```
iterator concurrent_vector::grow_by          ( size_t delta [ , const item_t & t ] );
iterator concurrent_vector::grow_to_at_least ( size_t n );
iterator concurrent_vector::push_back        ( const item_t & t );
```

Here, `item_t` corresponds to the template argument of `concurrent_vector`.

The function `grow_by()` will append `delta` new default constructed items (or copies of `t`) to the array. With `grow_to_at_least()`, the new array size is at least `n`. Finally, `push_back()` will append `t` to the end of the vector.

Remark

Consecutive items in a concurrent vector may not neccessarily have a consecutive address!

Dynamic Arrays

When growing a concurrent array, access to array items is still possible in parallel. However, new items may *not* yet have been constructed.

Therefore, when fetching items from an array, further tests are neccessary to ensure correctness when accessing item i:

- ensure, that the array size is at least i,
- ensure, that the element was constructed.

While the first test is simple, e.g. by using the `size()` function, to ensure that an object was constructed, is more involved.

For this, an `atomic` flag may be used as part of the object class, which is initialised last during construction. Also the value of the (atomic) element itself may signal construction, if it differs from some default value, e.g. 0.

The general procedure may look like:

```
item_t  fetch_item ( const concurrent_vector< item_t > &  v, const size_t i ) {
  while ( v.size() <= i )          // test vector size
    std::this_thread::yield();
  while ( ! v[i].is_constructed )  // test object construction
    std::this_thread::yield();
  return v[i];
}
```

Associative Arrays

An associative array, also called *hash map*, uses a *key* to access data. For this, a hash value of the key is computed and used as an index to some internal data structure (usually an array).

C++11 provides associative arrays as `unordered_map` and `unordered_multimap`. TBB extends these for concurrent access as *`concurrent_unordered_map`* and *`concurrent_unordered_multimap`*.

Both container types support concurrent insertion and traversal, but *not* concurrent deletion of elements.

The following functions are safe for concurrent access:

```
bool                        empty       () const;
size_t                      size        () const;

pair<iterator, bool>        insert      ( const value_t & x);
iterator                    insert      ( iterator hint, const value_t & x );
void                        insert      ( iterator first, iterator last );

iterator                    find        ( const key_t & k );
size_t                      count       ( const key_t & k ) const;
pair<iterator,iterator>     equal_range ( const key_t & k );
```

Furthermore, `concurrent_unordered_map` provides a multi-thread safe index operator `[]`.

Associative Arrays

Furthermore, safe iteration via *begin()* and *end()* is possible. The corresponding iterators remain valid, even if a new element is inserted.

All of these methods are *lock-free*, i.e. no user visible mutex is used.

TBB also implements *concurrent_hash_map*, which furthermore provides concurrent deletion, albeit with locks. Beside that, the interface is identical to the above classes.

An important difference affects iteration. It is *not* safe to use concurrent methods while iterating over a `concurrent_hash_map`.

Hashing

Hash values of keys are computed in TBB via the *tbb_hash* class

```
class tbb_hash {
  size_t operator () ( const key_t & k ) { return tbb_hasher( k ); }
};
```

For user defined keys, a new function `tbb_hasher()`

```
size_t tbb_hasher ( const key_t & k );
```

may be implemented to use the above container classes.

Associative Sets

Similar to associative arrays, C++11 defines associative sets which store elements accessed by a key value with no particular order, thereby enabling faster access.

Again, TBB extends these with concurrent insertion and traversal as *concurrent_unordered_set* and *concurrent_unordered_multiset*.

The interface is identical with the associative array counterparts, with the exception of an index operator.

Furthermore, hashing is identical to associative arrays, i.e. via `tbb_hasher` function.

Queues

The concurrent version of the C++ queue class in TBB is *concurrent_queue*.

It provides the basic functionality of a queue: appending items at its end and removing items from the top of the queue.

However, if multiple threads access a shared queue simultaneously, items at the top of the queue may no longer be available when trying to remove them.

Therefore, in TBB the `pop_front()` function is enhanced by a test for an item available. The resulting function is *try_pop()*:

C++ Sequential Access

```
std::queue< int >  q;
...
while ( ! q.empty() ) {
  int i = q.pop_front();
  ...
}
```

Parallel Access

```
concurrent_queue< int >  q;
...
int i;
while ( q.try_pop( i ) ) {
  ...
}
```

The return value of `try_pop()` is `true`, if an item was available, which is assigned to the function argument.

The counterpart of `try_pop()` is *push()*, which is equivalent to `push_back()` from `std::queue`.

Queues

Beside `push()` and `try_pop()`, `concurrent_queue` also has several functions equivalent to `std::queue` functions. But since these functions are not safe in a multi-threaded environment, they are prefixed with *`unsafe`*:

```
size_t   concurrent_queue::unsafe_size  () const;
iterator concurrent_queue::unsafe_begin ();
iterator concurrent_queue::unsafe_end   ();
```

While `concurrent_queue` will handle arbitrary many items, TBB also implements a *bounded* queue: *`concurrent_bounded_queue`*. Here, a maximal capacity may be specified, which is unlimited by default:

```
size_t concurrent_bounded_queue::capacity     () const;
void   concurrent_bounded_queue::set_capacity ( size_t capacity );
```

As a consequence, pushing items to the queue via `push()` may block until there is enough capacity to insert the new item. With *`try_push()`*, the push operation is skipped if it would exceed the queue capacity.

Furthermore, `concurrent_bounded_queue` implements a blocking *`pop()`* function, i.e. it will wait until the queue is non-empty. However, the function `try_pop()` is also available.

To wake up any thread waiting in `push()` or `pop()`, the *`abort()`* function can be used.

Queues

An Argument Against Queues

Since queues provide a FIFO access to data, insertion and removal has to follow a strict sequential order.

Remark

Due to concurrent insertion or removal, TBB only guarantees that if a single thread inserts two items and if another threads removes them, the order is preserved.

This sequential order may pose a bottleneck of many items are manages by a queue in short time.

Furthermore, due to the FIFO order, the time between data insertion and removal may be long, such that the data may have been removed from the local cache or, a remote thread may have pop'ed the data.

Therefore, it is suggested to use other functions instead of queues, e.g. `parallel_do` (with local insertion) or pipelines. Here, cache and thread locality is preserved.

Miscellanea

Miscellanea

TBB implementes various other algorithms, some of which are now available in C++11, e.g. atomics or timing.

Others are abbreviations of more general algorithms, e.g. `parallel_invoke()`, or implement standard algorithms, e.g. `parallel_sort()`.

TBB also provides a special memory allocator, which is more efficient than the default `malloc()` implementation in multi-threaded programs.

Finally, TBB handles C++ exceptions in a special way, simplifying error management.

Sorting

TBB provides a parallel sort algorithm *parallel_sort* which is based on *quick sort*:

```
template < typename iterator >
void parallel_sort ( iterator begin, iterator end );
```

Only random access iterators are supported in `parallel_sort`, e.g. for standard C arrays or `std::vector` but not for `std::list`:

```
std::vector< double >  x_vector( n );
std::list< double >    x_list( n );
...
parallel_sort( x_vector.begin(), x_vector.end() );
parallel_sort( x_list.begin(),   x_list.end() );   // error
```

The type `item_t` of the elements to sort has to provide a `swap` function:

```
void swap ( item_t & i1, item_t & i2 );
```

Remark

The quick sort implementation of `parallel_sort` is unstable as it may change the order of identical elements, but deterministic as it will always compute the same order for the same input set.

Sorting

By default, `parallel_sort` uses *`std::less`* to compare elements. For user defined types or if a special ordering should be used, a third argument may be provided to `parallel_sort`:

```
template < typename iterator, typename compare_t >
void parallel_sort ( iterator begin, iterator end,
                     const compare_t & compare );
```

Here, `compare_t` has to provide:

```
bool  compare_t::operator ()  ( const item_t & x, const item_t & y );
```

which should return `true` iff $x < y$, where $<$ is meant with respect to the corresponding ordering relation.

Sorting

By default, `parallel_sort` uses *`std::less`* to compare elements. For user defined types or if a special ordering should be used, a third argument may be provided to `parallel_sort`:

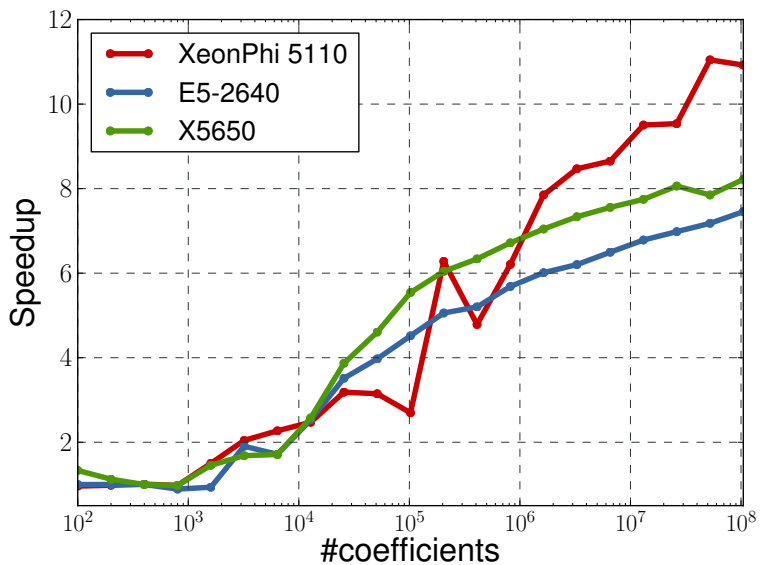
```
template < typename iterator, typename compare_t >
void parallel_sort ( iterator begin, iterator end,
                     const compare_t & compare );
```

Here, `compare_t` has to provide:

```
bool  compare_t::operator ()  ( const item_t & x, const item_t & y );
```

which should return `true` iff $x < y$, where $<$ is meant with respect to the corresponding ordering relation.

The speedup of `parallel_sort` depends on the number of elements as can be seen in the right diagram. But even for large sets, the speedup is not optimal (but increasing).



Parallel Functions

Having function objects `func0_t`,...,`func9_t` implementing

```
void func0_t::operator () ();
void func1_t::operator () ();
...
void func9_t::operator () ();
```

parallel execution of all functions can be accomplished by using the *parallel_invoke* algorithm:

```
template < typename func0_t,
           typename func1_t,
           ...,
           typename func9_t>
void parallel_invoke ( const func0_t & f0,
                       const func1_t & f1,
                       ...,
                       const func9_t & f9 );
```

The return values of all function objects is ignored and should default to `void`.

Remark

The `parallel_invoke` algorithm is implemented for two up to ten functions.

Parallel Functions

All functions are called by `parallel_invoke` *without* arguments. To execute functions with arguments in parallel, beside functors, lambda functions can be employed:

```
void f ( int );
void g ( double );

parallel_invoke( [] { f( 1   ); },
                 [] { g( 2.1 ); } );
```

This way, also multiple return values can be implemented:

```
int    f ( int );
double g ( double );
float  h ( float );

int     i;
double  d;
float   f;

parallel_invoke( [&i] { i = f( ... ); },
                 [&d] { d = g( ... ); },
                 [&f] { f = h( ... ); } );
```

Parallel Functions

Recursive execution of `parallel_invoke` is also possible:

```
void traverse_tree ( node_t & node ) {
  if ( ! node.is_leaf() ) {
    parallel_invoke( [&node] { traverse_tree( node.left  ); },
                     [&node] { traverse_tree( node.right ); },
  }

  handle( node );
}
```

Exceptions

For exceptions thrown within TBB algorithms or tasks, the following handling is implemented:

1. the exception is captured and all further exception with in the task group are ignored,
2. the task group is cancelled,
3. after the current algorithm has finished, the exception is rethrown by the thread which has executed the algorithm.

In order to rethrow the original exception in step 3, the program has to be compiled with C++11 support. Otherwise, the exception is of type *captured_exception*.

Assuming C++11, this enables error handling as for sequential programming:

```
void main () {
  ...
  try {
    parallel_for ( ... );
  }
  catch ( my_exception & e ) {
    ...
  }
}
```

Memory Allocation

TBB comes with an extra library "tbbmalloc", implementing optimised versions of `malloc()` and `free()`. The optimisations have a special focus for multi-threaded applications are concern

- scalability and
- false sharing.

To use it in applications, just add the library during linking:

```
> icpc -tbb -o myprog -c myprog.cc -ltbbmalloc
```

Furthermore, TBB implements several allocators following the C++ allocator requirements:

`tbb_allocator:`	uses the TBB versions of `malloc()` and `free()`,
`scalable_allocator:`	allocating and freeing memory scales with the number of processors,
`cache_aligned_allocator:`	allocate memory at cache line boundaries, preventing false sharing,
`zero_allocator:`	wrapper for other allocators, ensuring that the memory is zeroed after allocation.

Literature

"*Intel OpenCourseWare: Threading Building Blocks*". http://intel-software-academic-program.com/courses/.

"*Intel Threading Building Blocks Documentation*".
http://software.intel.com/sites/products/documentation/doclib/tbb_sa/help/index.htm.

"*Lambda functions*". http://en.cppreference.com/w/cpp/language/lambda.

"*Threading Building Blocks*". http://threadingbuildingblocks.org/.