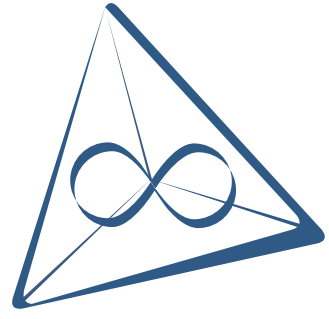
Task-Based $\mathcal{H}$-Matrix Arithmetics

Part II: Implementation

Ronald Kriemann
MPI MIS

Winterschool on $\mathcal{H}$-Matrices

2014



Threading Building Blocks

Threading Building Blocks

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 explicitly or implicitly define tasks in a parallel program. These tasks are then mapped by an internal scheduler to *worker threads*.

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

Threading Building Blocks

The standard introductory example looks as follows using TBB:

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

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

int main () {
  hello & t = * new( tbb::task::allocate_root() ) hello;
  tbb::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.

Threading Building Blocks

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 guarantee upon the execution time can be given.

Threading Building Blocks

TBB provides algorithms for loop parallelisation, reductions, sorting and pipelining.

These algorithms are based on the fundamental data structure of a

task

Furthermore, data structures for mutual exclusion and containers for concurrent access are implemented in TBB.

Finally, a special memory allocation library for multi-threaded programs comes along with TBB.

Lambda Functions

Lambda Functions

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

Beside the function body, a lambda function also contains a *data environment*, e.g. outside variables referenced inside the function body.

Lambda functions have the following form:

```
[ n, & p ]                  // captured outside variables
( int j )                   // function parameters  (optional)
-> void                     // return type          (optional)
{                           // function body
  p = j*n;
}
```

This can be simplified if no parameters are used or the return type can be determined automatically by the compiler:

```
[ n, & p ] { p = 5*n; }
```

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 *value* or by *reference*.

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

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

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
```

Both can be combined:

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

Functors

A typical application for lambda functions are *functors*:

```
//
// general function to traverse tree and apply functors
//
template < typename Functor >
void apply ( Tree * t, Functor & f ) {
  // apply f to current node
  f( t );

  // recurse
  if ( t->left  != nullptr ) apply( t->left );
  if ( t->right != nullptr ) apply( t->right );
}

Tree * root = build_tree();
int    sum  = 0;
int    prod = 1;

apply( root, [&sum]  ( Tree * t ) { sum  += t->value; } );
apply( root, [&prod] ( Tree * t ) { prod *= t->value; } );
apply( root, []      ( Tree * t ) { std::cout << t->value; } );
```

$\mathcal{H}$-Matrix Multiplication

$\mathcal{H}$-Matrix Multiplication

The algorithm for task-based $\mathcal{H}$-matrix multiplication is

```
procedure MULTIPLY(α, A, B, C)
    if A, B, C are block matrices then
        for i, j ∈ 0, 1 do
            task
                for ℓ ∈ 0, 1 do
                    MULTIPLY( α, A_ij, B_iℓ, C_ℓj );
    else
        task
            C := C + αA · B;
```

Here, *nested loop-parallelisation* is used to define the tasks in a hierarchical way.

Loop Parallelisation

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 );
```

Loop Parallelisation

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.

Furthermore, if the index set is not one dimensional, multiple invocations of `parallel_for` are needed.

Loop Parallelisation

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.

Furthermore, if the index set is not one dimensional, multiple invocations of `parallel_for` are needed.

For such cases, a generalised version of `parallel_for` based on *ranges* is available:

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

The *body* object must have the following interface:

```
void ( const range_t & r ) { ... }
```

Loop Parallelisation

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

TBB defines ranges for one, two and three dimensional index sets:

```
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;

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

Loop Parallelisation

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

```
blocked_range< size_t >    r1( 0, 100 );
blocked_range2d< size_t >  r2( 0, 100,  0, 10 );
blocked_range3d< size_t >  r3( 0, 100,  0, 20,  0, 5 );

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 )
      ...
```

Implementation

Using `parallel_for` together with `blocked_range2d`, the outer loops of the matrix multiplication are parallelised:

```
void multiply ( real alpha, TMatrix * A, TMatrix * B,
                TMatrix * C ) {
  if ( is_blocked( A, B, C ) ) {
    tbb::parallel_for(

      tbb::blocked_range2d< uint >( 0, C->nblock_rows(),
                                    0, C->nblock_cols() ),

      [alpha,A,B,C] ( const tbb::blocked_range2d< uint > & r ) {
        for ( auto i = r.rows().begin(); i != r.rows().end(); ++i )
          for ( auto j = r.cols().begin(); j != r.cols().end(); ++j )
            for ( int l = 0; l < A->nblock_cols(); ++l )
              multiply( alpha, A->block(i,l), B->block(l,j),
                        C->block(i,j) );
    } );
  }
  else {
    multiply_leaf( alpha, A, B, C );
} }
```

Recursive $\mathcal{H}$-LU Factorisation

Recursive $\mathcal{H}$-LU Factorisation

For an $\mathcal{H}$-matrix A with a $n \times n$ block structure, the LU factorisation algorithm is:

```
procedure LU(A)
    for 0 ≤ i < n  do
        LU( A_ii );                                   // Recursion

        for i < j < n  do                             // Solve in Row/Column
            SolveLower(A_ii, A_ij );
            SolveUpper(A_ii, A_ji );

        for i < ℓ < n  do                             // Update Trailing Sub Matrix
            for i < j < n  do
                Multiply(−1, A_ℓi, A_ij, A_ℓj );
```

Again, nested loop parallelisation can be applied. Furthermore, matrix solves can be done in parallel.

Parallel Invocation

Up to ten functions can be directly executed in parallel by the TBB function *parallel_invoke*:

```
tbb::parallel_invoke( f0, f1 );
tbb::parallel_invoke( f0, f1, f2, f3 );
tbb::parallel_invoke( f0, f1, f2, f3, ..., f9 );
```

However, two limitations exist for the called functions:

- no arguments and
- no return value.

Both can be overcome with lambda functions:

```
tbb::parallel_invoke( [&i,n] { i = f( n ); },
                      [&d,e] { d = g( e ); },
                      [&f,x] { f = h( x ); } );
```

Implementation

The $\mathcal{H}$-LU implementation uses standard and range based `parallel_for` together with `parallel_invoke`:

```
void factorise ( TMatrix * A ) {
  for ( int i = 0; i < A->nblock_rows(); ++i ) {
    factorise( A->block(i,i) );

    tbb::parallel_invoke(
      [A,i] { tbb::parallel_for( i+1, A->nblock_rows(), solve_upper ); },
      [A,i] { tbb::parallel_for( i+1, A->nblock_cols(), solve_lower ); }
    );

    tbb::parallel_for(

      tbb::blocked_range2d< int >( i+1, A->nblock_rows(),
                                   i+1, A->nblock_cols() ),

      [A,i] ( const tbb::blocked_range2d< int > &  r )
        for ( auto l = r.rows().begin(); l != r.rows().end(); l++ )
          for ( auto j = r.cols().begin(); j != r.cols().end(); j++ )
            multiply( -1, A->block(l,i), A->block(i,j), A->block(l,j) );
    );
  } }
```

$\mathcal{H}$-Matrix Construction

$\mathcal{H}$-Matrix Construction

$\mathcal{H}$-matrix construction with coarsening uses explicit tasks with task dependencies:

procedure $\textsc{MatrixConstruct}(b \in T)$
  if $b \in \mathcal{L}(T)$ **then**
    task
      build leaf matrix;
  else
    task
      for all $b' \in \mathcal{S}(b)$ **do**
        sub task: $\textsc{MatrixConstruct}(b')$;
      coarsen matrix for b;

Tasks

Explicit tasks in TBB are objects of classes derived from *task*:

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

The member function *execute* has to be overwritten with the actual algorithm of the task

```
class mytask : public tbb::task {
  public:
  task * execute() {
    // do step 1
    ...
    // do step 2
    ...
  }
};
```

Tasks

New *child* tasks are spawned from a *parent* task using the function spawn:

```
class mytask : public tbb::task {
  task * execute() {
    myclass & t = * new ( ... ) myclass;  // create child task
    spawn( t );                           // spawn child task
    ...
} };
```

A task without a parent is a *root* task.

To synchronise with the end of child tasks, use wait_for_all:

```
class mytask : public tbb::task {
  task * execute() {
    myclass & t = * new ( ... ) myclass;
    spawn( t );
    wait_for_all();                 // wait for all child tasks
    ...
} };
```

Reference Counting

For task synchronisation to work in TBB, each task has a *reference counter* which must equal the number of child tasks *before* any child task is spawned:

```
class mytask : public tbb::task {
  task * execute() {
    myclass & t = * new ( ... ) myclass;
    set_ref_count( 2 );           // 1 plus 1 extra for "wait"
    spawn( t );
    wait_for_all();               // wait until refcount is 1
    ...
} };
```

The functions spawn and wait_for_all can also be combined:

```
class mytask : public tbb::task {
  task * execute() {
    myclass & t = * new ( ... ) myclass;
    set_ref_count( 2 );
    spawn_and_wait_for_all( t );
    ...
} };
```

Task Lists and Root Tasks

Instead of spawning single tasks, *task lists* are available:

```
class mytask : public tbb::task {
  task * execute() {
    myclass &      t1 = * new ( ... ) myclass;
    myclass &      t2 = * new ( ... ) myclass;
    tbb::task_list t;

    t.push_back( t1 );
    t.push_back( t2 );

    set_ref_count( 3 );
    spawn_and_wait_for_all( t );
    ...
} };
```

For root tasks, e.g. without a parent task, a special spawn function is provided:

```
myclass & root = * new ( ... ) myclass;

spawn_root_and_wait_for_all( root );
```

Implementation

```
class build_task_t : public tbb::task {
  tbb::task *  execute () {
    if ( bc->is_leaf() ) {
      return build_leaf( bc );
    } else {
      tbb::task_list  subtasks;

      for ( int j = 0; j < bc->ncols(); ++j )
        for ( int i = 0; i < bc->nrows(); ++i )
          subtasks.push_back( *new build_task_t( bc->son( i,j ) ) );

      set_ref_count( bc->ncols()*bc->nrows()+1 );
      spawn_and_wait_for_all( subtasks );

      M = new TBlockMatrix( ... );
      coarsen( M );
    }
    return nullptr;
  }
};
build_task_t & t = *new build_task_t( root );
tbb::task::spawn_root_and_wait( t );
```

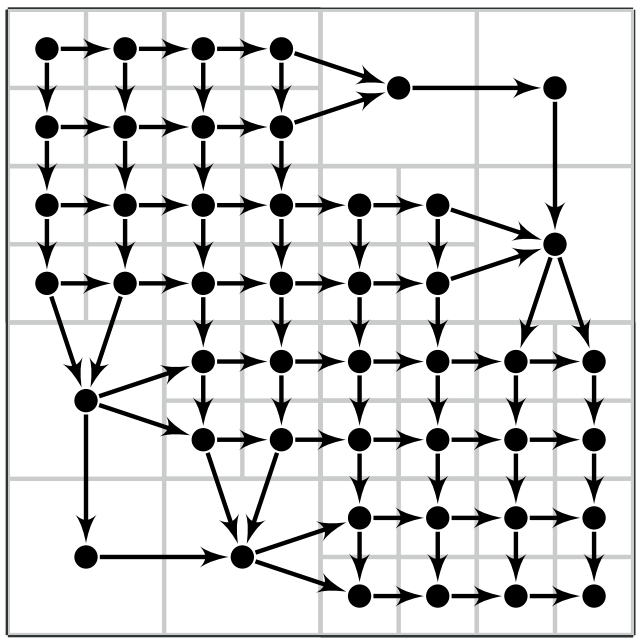
Task-based $\mathcal{H}$-LU Factorisation

Task-based $\mathcal{H}$-LU Factorisation

A DAG computation decomposes into two steps:

1. construct the DAG and
2. execute the DAG.

Since each task is explicitly constructed *before* DAG execution, all tasks are root tasks. Therefore, successor tasks have to be spawned *manually*.



DAG Construction

```
fac_node_t * build_dag ( TMatrix * A ) {
  for ( int i = 0; i < A->nblock_rows(); ++i ) {
    // factorisation task
    fac_node_t *  diag_node = build_dag( A_ii );

    // solve tasks
    for ( auto M = column_i_start; M != column_i->end(); ++M ) {
      solve_U_node_t *  solve_node = new solve_U_node_t( M, A->block(i,i) );
      solve_node->add_dep( diag_node ); // solve after factorisation
    }
    for ( auto M = row_i_start; M != row_i->end(); ++M ) { ... }

    // update tasks
    for ( auto U = row_i_start; U != row_i->end(); ++U ) {
      for ( auto L = column_i_start; L != column_i->end(); ++L ) {
        // destination block of update
        TMatrix * M = H->block( L->row_is(), U->col_is() );
        // only update if one is a leaf block
        if ( is_leaf( A->block(i,i), U, L, M ) ) {
          update_node_t *  upd_node = new update_node_t( M, L, U );

          upd_node->add_rec_dep( U ); // update after solve
          upd_node->add_rec_dep( L );
} } } } }
```

Storing successor tasks also has to be managed manually.

Factorisation Task

For dependency tests for successor tasks, the reference counting of TBB tasks may be used:

```
class fac_task_t : public tbb::task {
  public:
  tbb::task execute  () {
    // factorise leaf blocks
    if ( is_leaf( _A ) ) {
      LU::factorise_dense( _A );
    }

    // spawn successors
    for ( auto  node : _successors ) {
      if ( node->decrement_ref_count() == 0 ) {
        spawn( * node );
      }
    }

    return nullptr;
  }
};
```

DAG Execution

The DAG is executed beginning with the start nodes of the DAG.

The end node of the DAG is *not* executed by TBB but only waited for until all dependencies are met. Task execution *and* destruction is performed manually.

```
tbb::task *     final;
tbb::task_list  start;

build_dag( A, final, start, ... );

final->increment_ref_count();             // one extra for wait
final->spawn_and_wait_for_all( start );

final->execute();                         // execute end node
tbb::task::destroy( * final );            // destroy end node
```

Literature

Literature

- R. Kriemann,
  Parallel Programming,
  http://www.kriemann.name/Ronald/.
- *Threading Building Blocks*,
  http://threadingbuildingblocks.org.
- *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