C++11 has Thread Support

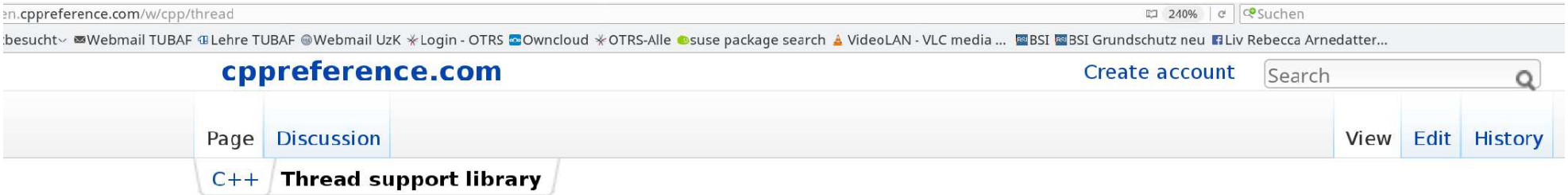
http://en.cppreference.com/w/cpp/thread

However, C++11 (2011!) support is still experimental in g++ 4.8.5 (2015; in use in current stable Linux distros still 2017; gcc 7 (2019) has full C++14 support) and has to be manually turned on using `-std=c++11`:

```
/usr/include/c++/4.8/bits/c++0x_warning.h:32:2:
error: #error This file requires compiler and library support
for the ISO C++ 2011 standard. This support is currently experimental,
and must be enabled with the -std=c++11 or -std=gnu++11 compiler options.
```

The GNU implementation of C++11 threads uses the pthreads library, and therefore the pthreads-library has to be linked!

- The design is completely different from OpenMP, e.g., does not have a simple parallel loop.
- The task-based paradigm is designed after Pthreads (and Boost threads).
- The STL is still not thread safe :(
- However, parallel reads are O.K. in STL containers; parallel writes of different objects O.K.;
- Write conflicts (race conditions) need something like a “critical” in OpenMP;
  in C++11 this functionality is provided by the `mutex` (mutually exclusive) and `Lock` class;



Thread support library

C++ includes built-in support for threads, mutual exclusion, condition variables, and futures.

Threads

Threads enable programs to execute across several processor cores.

Defined in header <thread>

thread (C++11)	manages a separate thread (class)

Functions managing the current thread

Defined in namespace this_thread

yield (C++11)	suggests that the implementation reschedule execution of threads (function)
get_id (C++11)	returns the thread id of the current thread (function)
sleep_for (C++11)	stops the execution of the current thread for a specified time duration (function)
sleep_until (C++11)	stops the execution of the current thread until a specified time point (function)

Cache size access

Defined in header <new>

hardware_destructive_interference_size	(C++17)	min offset to avoid false sharing
hardware_constructive_interference_size		max offset to promote true sharing (constant)

Simple Example

```
#include <iostream>
#include <vector>
#include <thread>
// needs: g++ -std=c++11 -pthread

using namespace std;
void testme(int N)
{
  cout<<"Thread: "<<this_thread::get_id()<<"  N ist "<<N<<endl;
}

int main()
{
  thread t1(testme,2);  // launches an additional thread which executes testme
  thread t2(testme,3);  // and another one
  thread t3(testme,4);  // and another one
  testme(1)             // the master thread can of course also do something

  t1.join(); t2.join(); t3.join(); // joins threads back to the master (=OpenMP)
}
```

Output:

```
> g++ -g -std=c++11 -pthread  cpp14_test_threads.cpp
> ./a.out
Thread: 139925282436928  N ist 1
Thread: 139925265450752  N ist 2
Thread: 139925257058048  N ist 3
Thread: 139925248665344  N ist 4
```

"`-std=c++11`" switches on C++11 support
"`-pthread`" links pthread lib

Another Example

```
#include <iostream>
#include <thread>
#include <chrono>
#include <vector>
// needs: g++ -std=c++11 -pthread
using namespace std;

void multiply(const int N, const vector<vector<double> > &a,
              const vector<vector<double> > &b, vector<vector<double> > &c) {
  int i,j,k;
  cout<<"Multiply Thread: "<<this_thread::get_id()<<endl;
  for (i = 0; i < N; i++) {
      for (j = 0; j < N; j++) {
          c[i][j] = 0;
          for (k = 0; k < N; k++) {
             c.at(i).at(j) += a.at(i).at(k) * b.at(k).at(j);
          }
      }
  }
}
```

```
int main()
{
  int i,j,k;

  vector<vector<double> > a(N);
  vector<vector<double> > b(N);
  vector<vector<double> > c(N);

  for(int i=0; i<N; i++) {
      a.at(i).resize(N);
      b.at(i).resize(N);
      c.at(i).resize(N);
  }

  for (i = 0; i < N; i++) {       // fill matrices
      for (j = 0; j < N; j++) {
          a.at(i).at(j) = i+j+1.0;
          b.at(i).at(j) = 1.0/(i+j+1.0);
          c.at(i).at(j) = 0.0;
      }
  }
```

```
vector<vector<double> > a1(a); // copy data
vector<vector<double> > b1(b);
vector<vector<double> > c1(c);

vector<vector<double> > a2(a); // copy data
vector<vector<double> > b2(b);
vector<vector<double> > c2(c);

{
  testme(0);
  thread t1(testme, 1); // spawn a test thread
  t1.join();            // join in back to master
}
```

```
  {
    // now spawn two parallel threads
    // we use different matrices for every thread here!

    thread t1(multiply, N,std::ref(a1),std::ref(b1),std::ref(c1));
    thread t2(multiply, N,std::ref(a2),std::ref(b2),std::ref(c2));

    //multiply(N,a,b,c); // master thread can also do work in parallel


    t1.join(); t2.join();
  }
}
```

Output:

```
Thread: 140281508857664  N ist 0
Thread: 140281491871488  N ist 1
Multiply Thread: 140281483478784
Multiply Thread: 140281491871488
```

Launching a Group of Threads

```
#include <iostream>
#include <thread>
static const int num_threads = 10;
void func(int tid) {
    std::cout << "Called by thread " << tid << std::endl;
}
void main() {
    std::thread t[num_threads]; //An array of threads
    // std::vector<std::thread> t(num_threads); // or a vector :)

    //Launch a group of threads
    for (int i = 0; i < num_threads; ++i) {
        t[i] = std::thread(func, i);
    }
    std::cout << "Launched from the main\n";
    //Join the threads with the main thread
    for (int i = 0; i < num_threads; ++i) {
        t[i].join();
    }
}
```

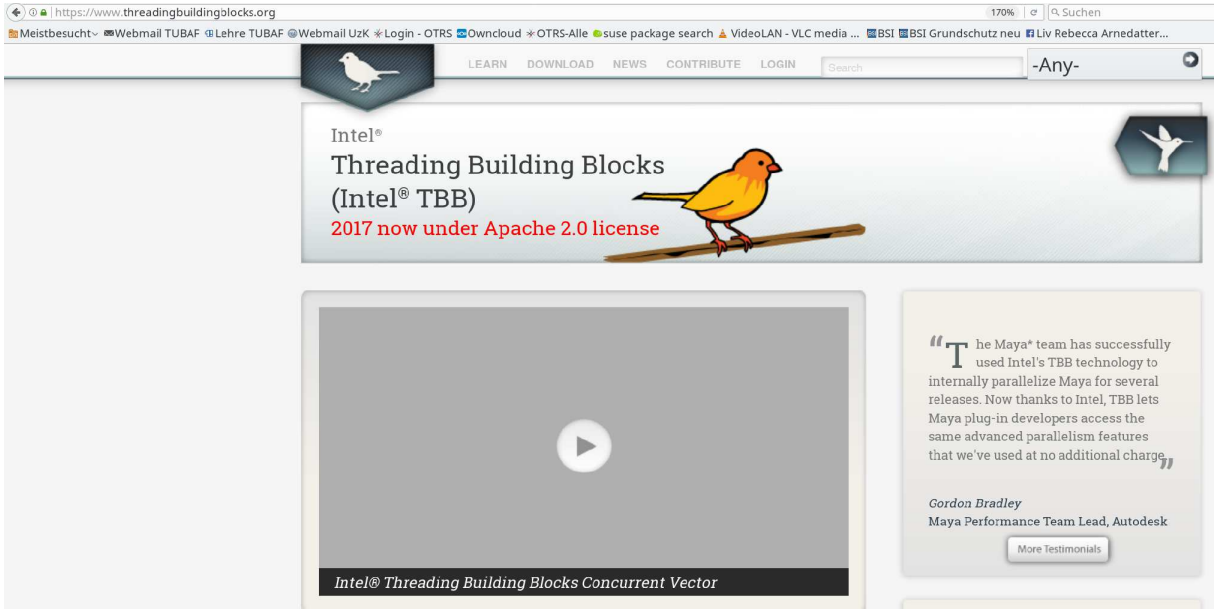
Example using std::async from cppreference.com

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <future>
template <typename RAIter>
int parallel_sum(RAIter beg, RAIter end) {
    auto len = end - beg;
    if(len < 1000)
        return std::accumulate(beg, end, 0);
    RAIter mid = beg + len/2;
    auto handle = std::async(std::launch::async,
                             parallel_sum<RAIter>, mid, end);
    int sum = parallel_sum(beg, mid);
    return sum + handle.get();
}
int main() {
    std::vector<int> v(10000, 1);
    std::cout << "The sum is " << parallel_sum(v.begin(), v.end()) << '\n';
}
```

For serious parallel algorithms Intel TBB is still much better

TBB = Threads Building Blocks

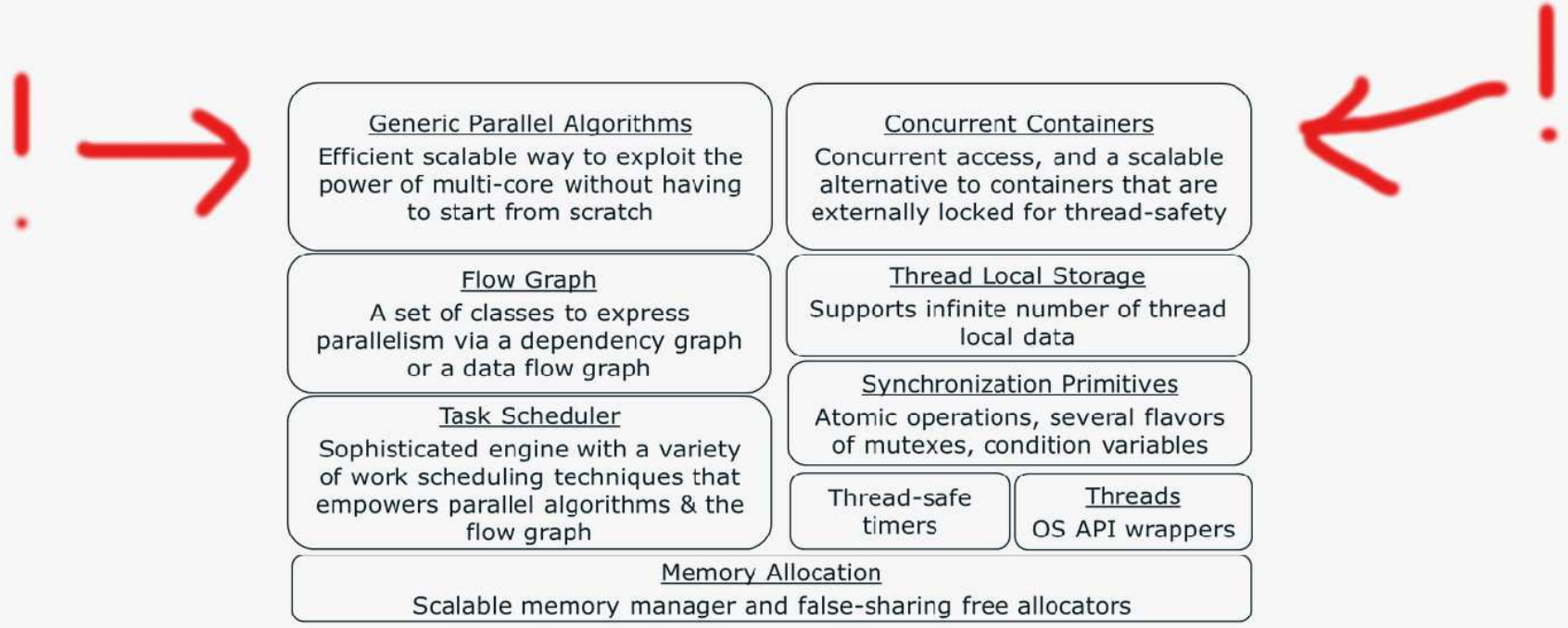
- First version introduced by Intel Corp. in 2006 (!) but is now open (since 2017 under Apache license).
- Threads appeared in the C++ standard only in C++11 (2011)!
- Official Website: `https://www.threadingbuildingblocks.org/`
- Provides serious support for parallel algorithms (unlike C++14)!
- The following is from https://www.threadingbuildingblocks.org/intel-tbb-tutorial



TBB has what OpenMP programmers miss in C++14

Learn about the Intel® Threading Building Blocks library

The Intel® Threading Building Blocks (Intel® TBB) library provides software developers with a solution for enabling parallelism in C++ applications and libraries. The well-known advantage of the Intel TBB library is that it makes parallel performance and scalability easily accessible to software developers writing loop- and task-based applications. The library includes a number of generic parallel algorithms, concurrent containers, support for dependency and data flow graphs, thread local storage, a work-stealing task scheduler for task based programming, synchronization primitives, a scalable memory allocator and more.



TBB has what OpenMP programmers miss in C++14

Has a parallel for loops (like OpenMP) and (more general!) parallel do loops:

```
#include "tbb/tbb.h"
using namespace tbb;

void ParallelApplyFoo(float a[], size_t n)
  {
    tbb::parallel_for(size_t(0), n, [&](size_t i) { Foo(a[i]); } );
  }

// example uses Lambda function from C++11
// http://de.cppreference.com/w/cpp/language/lambda
// (but can also be used without)
```

Excursion: C++11 Lambda Functions

Similar to anonymous functions in Matlab/Octave:

```
% Matlab/Octave:
 >> func=@(x) x+4;
 >> func(6)
ans =  10
```

```
// TBB + C++11:
#include <vector>
#include <iostream>
#include <algorithm>
#include <functional>

int main()  // needs -std=c++11
{
    std::function<int (int)> func = [](int i) { return i+4; };
    std::cout << "func: " << func(6) << '\n';
}
```

Excursion: C++11 Lambda Functions (2)

Especially usefull as an argument to generic algorithms (here `remove_if`).

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <functional>
int main()  // needs -std=c++11
{
  std::vector<int> c {1,2,3,4,5,6,7};
  int x=5;
  c.erase(std::remove_if(c.begin(),c.end(),[x](int n){ return n < x; } ),c.end());
  std::cout << "c: ";
  for(auto i: c) {
    std::cout << i << ' ';
  }
  std::cout << '\n';
}
```

Therefore also used in TBB for generic parallel algorithms.

TBB has thread save containers!! - unlike C++14

```
#include <vector>
#include "tbb/tbb.h"
int main(){
 std::vector<int> v; //STL vector - not thread safe, e.g., for size change
 tbb::concurrent_vector<int> w;   //TBB vector - thread safe even for push_back()!
}
```

Intel guarantees (https://software.intel.com/en-us/node/506203):
A `concurrent_vector` *is a container with the following features:*

- *Random access by index. The index of the first element is zero.*
- *Multiple threads can grow the container and append new elements concurrently.*
- *Growing the container does not invalidate existing iterators or indices.*

and *"Unlike a std::vector, a* `concurrent_vector` **never moves existing elements when it grows**. *The container* **allocates a series of contiguous arrays**. *The first reservation, growth, or assignment operation determines the size of the first array. Using a small number of elements as initial size incurs fragmentation across cache lines that may increase element access time. The method* `shrink_to_fit()` *merges several smaller arrays into a single contiguous array, which may improve access time."*

TBB has thread save containers!! - unlike C++14 (2)

- Much more convenient than STL containers for parallel algorithms.
- A TBB `concurrent_vector` will not move any existing element when the `concurrent_vector` grows: all references and pointers will stay valid.
- An STL `vector` will move existing elements $\log_2(n)$ times during growth of the vector; adds overhead during growth but no overhead when accessing using `operator[]` .
- Therefore, STL vectors are considered as fast as C/C++ builtin arrays and can (=should) replace C/C++ builtin arrays routinely everywere.
- Of course the TBB strategy will add overhead for the internal management of the memory blocks for every access using `operator[]`
- Also, unlike in a std::vector, the iterators are not raw pointers.
- For parallel algorithms the benefits of TBB `concurrent_vector` are much more important: It is better to compute the correct result than computing an incorrect result (due to a race condition) fast!