Intel Threading Building Blocks

Outfitting C++ for Multi-core Processor Parallelism

Michael Müller

DEUTSCHES
KREBSFORSCHUNGSZENTRUM
IN DER HELMHOLTZ-GEMEINSCHAFT

Another threading library, why?

→Do not use threads, **automatically** map logical parallelism onto threads in a way that makes efficient use of processor resources

→Use pure C++ generic programming

→Compiler independence, processor independence, OS independence

Facts on TBB

- C++ Library for parallel computing
- Cross platform
- V4.0 since November 2011
- GPLv2 licensed, commercially available
- http://threadingbuildingblocks.org/

Some more details

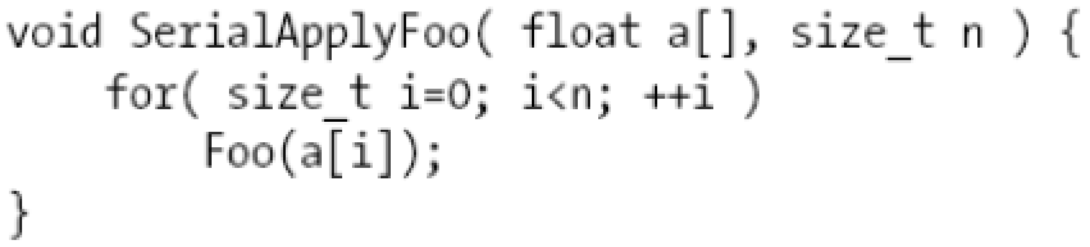
- TBB implements "task stealing" to balance a parallel workload
- TBB, like the STL, uses templates extensively
- TBB implements the pipeline pattern

Library components

- Basic algorithms: parallel_for, parallel_reduce, parallel_scan
- Advanced algorithms: parallel_while, parallel_do, parallel_pipeline, parallel_sort
- Containers: concurrent_queue, concurrent_vector, concurrent_hash_map
- Scalable memory allocation: scalable_malloc, scalable_free, scalable_realloc, scalable_calloc, scalable_allocator, cache_aligned_allocator
- Mutual exclusion: mutex, spin_mutex, queuing_mutex, spin_rw_mutex, queuing_rw_mutex, recursive mutex
- Atomic operations: fetch_and_add, fetch_and_increment, fetch_and_decrement, compare_and_swap, fetch_and_store
- Timing: portable fine grained global time stamp
- Task Scheduler: direct access to control the creation and activation of tasks

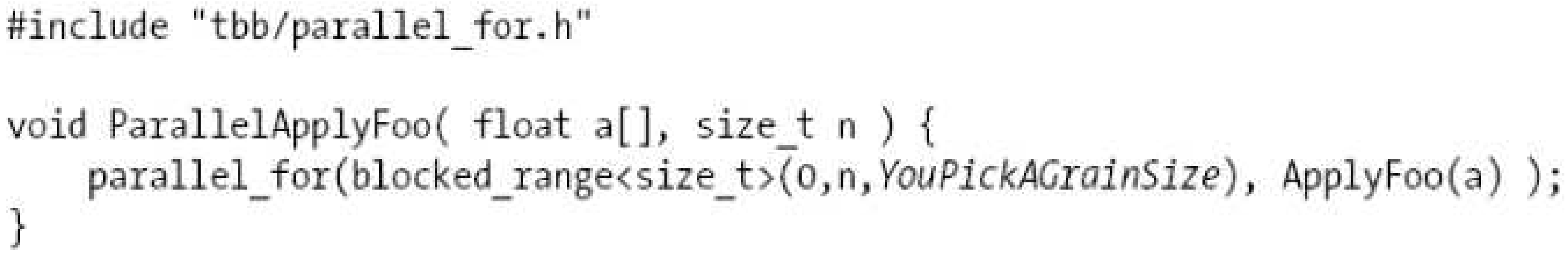
Code Example (I)

Serial:



```
void SerialApplyFoo( float a[], size_t n ) {
    for( size_t i=0; i<n; ++i )
        Foo(a[i]);
}
```

Parallel:

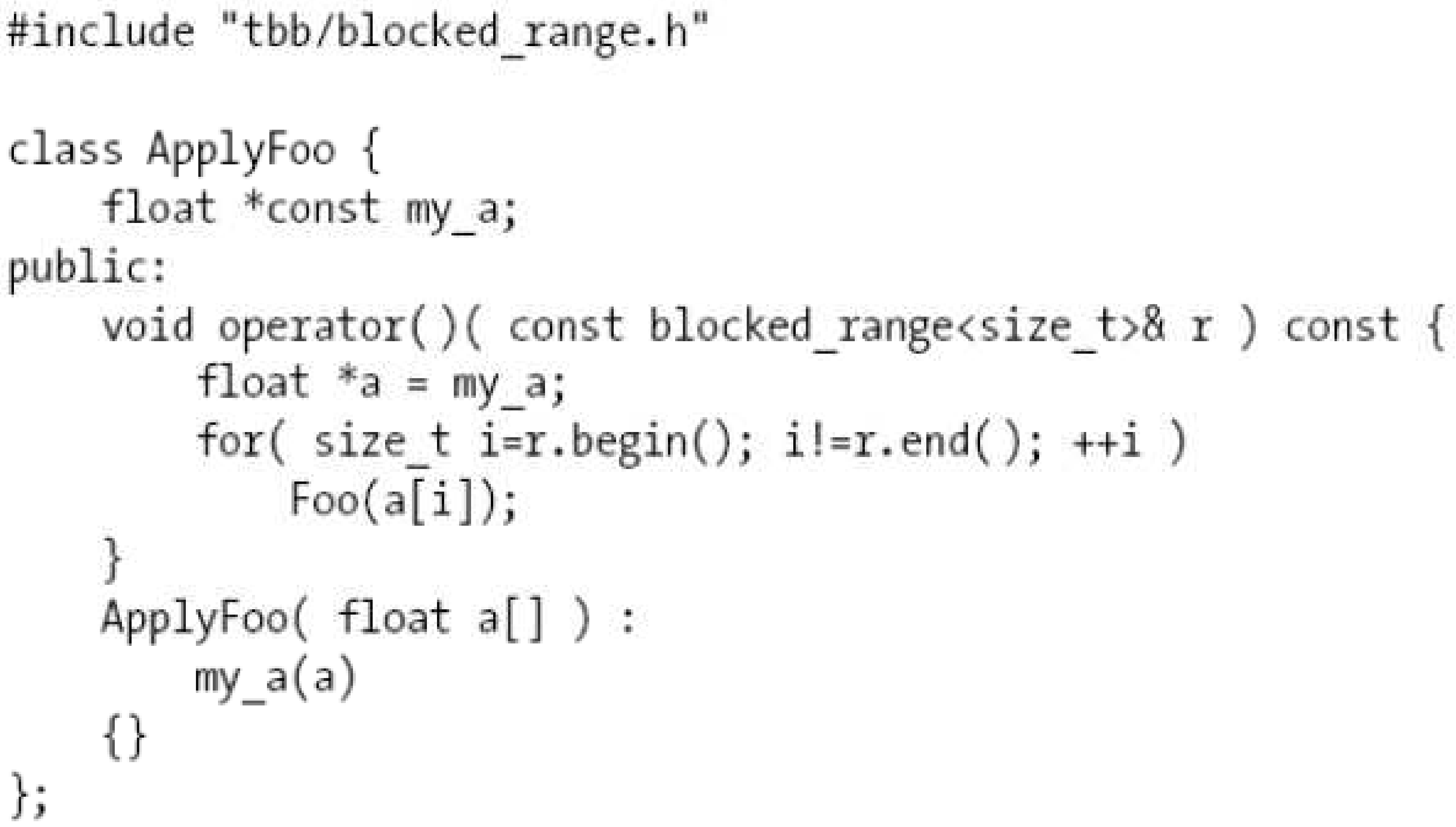


```
#include "tbb/parallel_for.h"

void ParallelApplyFoo( float a[], size_t n ) {
    parallel_for(blocked_range<size_t>(0,n,YouPickAGrainSize), ApplyFoo(a) );
}
```

Code Example (II)

Using functors:



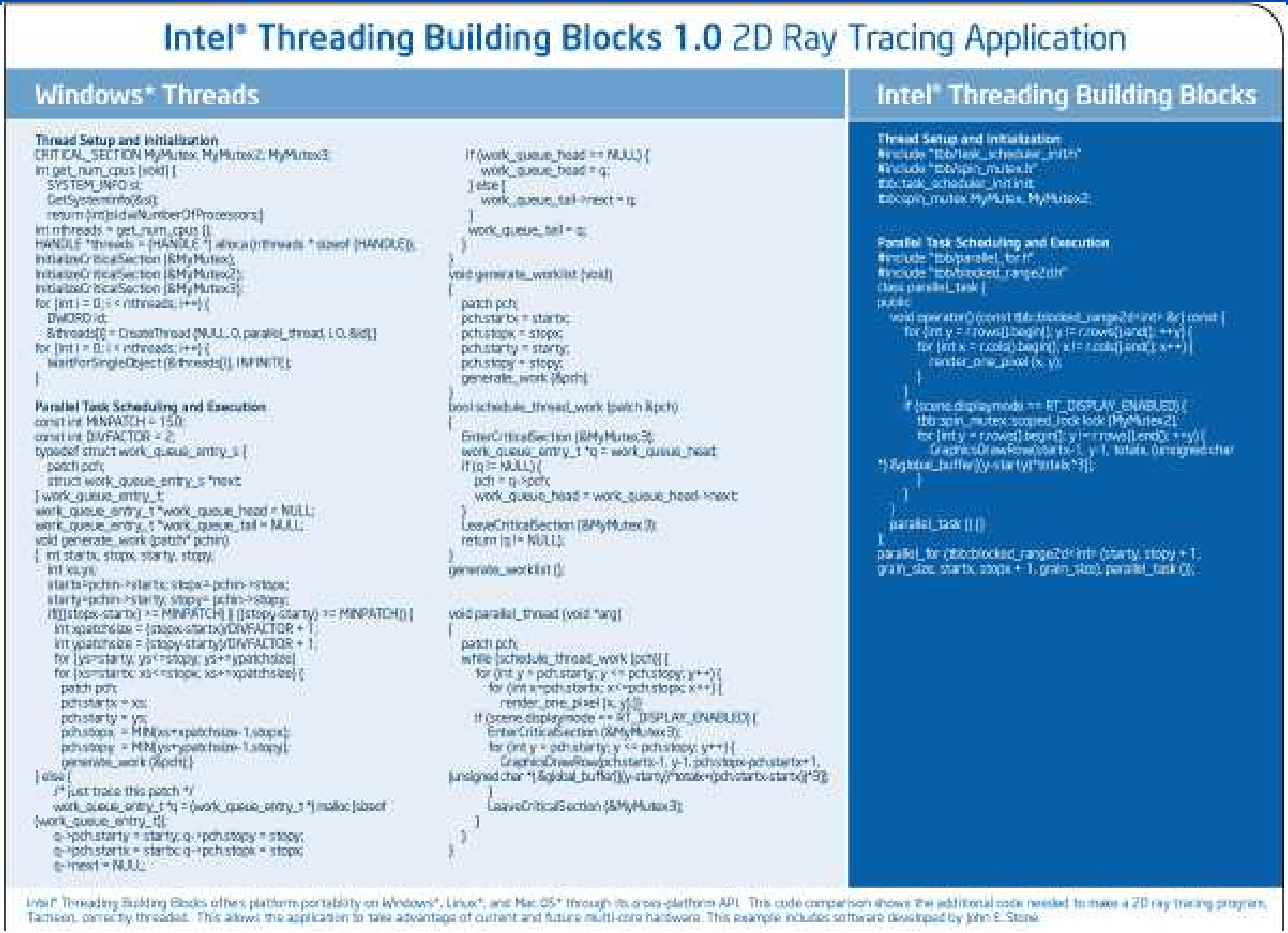
```
#include "tbb/blocked_range.h"

class ApplyFoo {
    float *const my_a;
public:
    void operator()( const blocked_range<size_t>& r ) const {
        float *a = my_a;
        for( size_t i=r.begin(); i!=r.end(); ++i )
            Foo(a[i]);
    }
    ApplyFoo( float a[] ) :
        my_a(a)
    {}
};
```

Code Example (III)

```
#include <tbb/task_scheduler_init.h>
int main() {
  tbb::task_scheduler_init init;
  // use of an algorithm
  ParallelApplyFoo( … )
}
```

TBB vs. Windows threads



Intel® Threading Building Blocks 1.0 2D Ray Tracing Application

Windows* Threads

Thread Setup and Initialization

```
CRITICAL_SECTION MyMutex, MyMutex2, MyMutex3;
int get_num_cpus (void) {
   SYSTEM_INFO si;
   GetSystemInfo(&si);
   return (int)si.dwNumberOfProcessors;}
int nthreads = get_num_cpus ();
HANDLE *threads = (HANDLE *) alloca (nthreads * sizeof (HANDLE));
InitializeCriticalSection (&MyMutex);
InitializeCriticalSection (&MyMutex2);
InitializeCriticalSection (&MyMutex3);
for (int i = 0; i < nthreads; i++) {
   DWORD id;
   &threads[i] = CreateThread (NULL, 0, parallel_thread, i, 0, &id);}
for (int i = 0; i < nthreads; i++) {
   WaitForSingleObject (&threads[i], INFINITE);
}
```

Parallel Task Scheduling and Execution

```
const int MINPATCH = 150;
const int DIVFACTOR = 2;
typedef struct work_queue_entry_s {
   patch pch;
   struct work_queue_entry_s *next;
} work_queue_entry_t;
work_queue_entry_t *work_queue_head = NULL;
work_queue_entry_t *work_queue_tail = NULL;
void generate_work (patch* pchin)
{  int startx, stopx, starty, stopy;
   int xs,ys;
   startx=pchin->startx; stopx= pchin->stopx;
   starty=pchin->starty; stopy= pchin->stopy;
   if(((stopx-startx) >= MINPATCH) || ((stopy-starty) >= MINPATCH)) {
     int xpatchsize = (stopx-startx)/DIVFACTOR + 1;
     int ypatchsize = (stopy-starty)/DIVFACTOR + 1;
     for (ys=starty; ys<=stopy; ys+=ypatchsize)
     for (xs=startx; xs<=stopx; xs+=xpatchsize) {
       patch pch;
       pch.startx = xs;
       pch.starty = ys;
       pch.stopx  = MIN(xs+xpatchsize-1,stopx);
       pch.stopy  = MIN(ys+ypatchsize-1,stopy);
       generate_work (&pch);}
} else {
     /* just trace this patch */
     work_queue_entry_t *q = (work_queue_entry_t *) malloc (sizeof
(work_queue_entry_t));
     q->pch.starty = starty; q->pch.stopy = stopy;
     q->pch.startx = startx; q->pch.stopx = stopx;
     q->next = NULL;
      if (work_queue_head == NULL) {
        work_queue_head = q;
      } else {
        work_queue_tail->next = q;
      }
      work_queue_tail = q;
    }
}
void generate_worklist (void)
{
   patch pch;
   pch.startx = startx;
   pch.stopx = stopx;
   pch.starty = starty;
   pch.stopy = stopy;
   generate_work (&pch);
}
bool schedule_thread_work (patch &pch)
{
   EnterCriticalSection (&MyMutex3);
   work_queue_entry_t *q = work_queue_head;
   if (q != NULL) {
     pch = q->pch;
     work_queue_head = work_queue_head->next;
   }
   LeaveCriticalSection (&MyMutex3);
   return (q != NULL);
}
generate_worklist ();

void parallel_thread (void *arg)
{
   patch pch;
   while (schedule_thread_work (pch)) {
     for (int y = pch.starty; y <= pch.stopy; y++) {
       for (int x=pch.startx; x<=pch.stopx; x++) {
         render_one_pixel (x, y);}}
     if (scene.displaymode == RT_DISPLAY_ENABLED) {
       EnterCriticalSection (&MyMutex3);
       for (int y = pch.starty; y <= pch.stopy; y++) {
         GraphicsDrawRow(pch.startx-1, y-1, pch.stopx-pch.startx+1,
(unsigned char *) &global_buffer[((y-starty)*totalx+(pch.startx-startx))*3]);
       }
       LeaveCriticalSection (&MyMutex3);
     }
   }
}
```

Intel® Threading Building Blocks

Thread Setup and Initialization

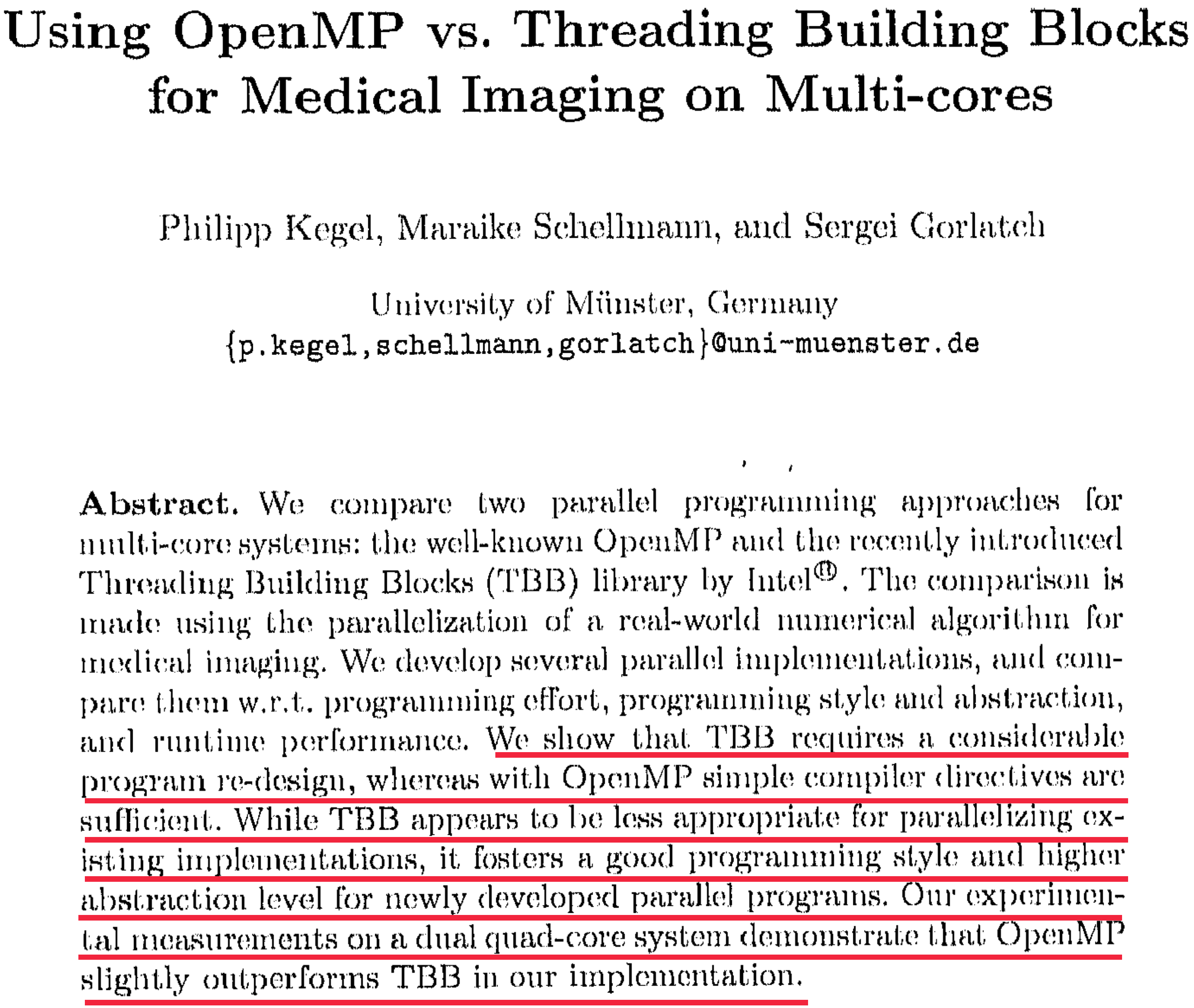
```
#include "tbb/task_scheduler_init.h"
#include "tbb/spin_mutex.h"
tbb::task_scheduler_init init;
tbb::spin_mutex MyMutex, MyMutex2;
```

Parallel Task Scheduling and Execution

```
#include "tbb/parallel_for.h"
#include "tbb/blocked_range2d.h"
class parallel_task {
public:
  void operator() (const tbb::blocked_range2d<int> &r) const {
    for (int y = r.rows().begin(); y != r.rows().end(); ++y) {
      for (int x = r.cols().begin(); x != r.cols().end(); x++) {
        render_one_pixel (x, y);
      }
    }
    if (scene.displaymode == RT_DISPLAY_ENABLED) {
      tbb::spin_mutex::scoped_lock lock (MyMutex2);
      for (int y = r.rows().begin(); y != r.rows().end(); ++y) {
        GraphicsDrawRow(startx-1, y-1, totalx, (unsigned char
*) &global_buffer[((y-starty)*totalx)*3]);
      }
    }
  }
  parallel_task () {}
};
parallel_for (tbb::blocked_range2d<int> (starty, stopy + 1,
grain_size, startx, stopx + 1, grain_size), parallel_task ());
```

Intel® Threading Building Blocks offers platform portability on Windows*, Linux*, and Mac OS* through its cross-platform API. This code comparison shows the additional code needed to make a 2D ray tracing program, Tachyon, correctly threaded. This allows the application to take advantage of current and future multi-core hardware. This example includes software developed by John E. Stone.

TBB vs. OpenMP



Using OpenMP vs. Threading Building Blocks for Medical Imaging on Multi-cores

Philipp Kegel, Maraike Schellmann, and Sergei Gorlatch

University of Münster, Germany
{p.kegel,schellmann,gorlatch}@uni-muenster.de

Abstract. We compare two parallel programming approaches for multi-core systems: the well-known OpenMP and the recently introduced Threading Building Blocks (TBB) library by Intel®. The comparison is made using the parallelization of a real-world numerical algorithm for medical imaging. We develop several parallel implementations, and compare them w.r.t. programming effort, programming style and abstraction, and runtime performance. We show that TBB requires a considerable program re-design, whereas with OpenMP simple compiler directives are sufficient. While TBB appears to be less appropriate for parallelizing existing implementations, it fosters a good programming style and higher abstraction level for newly developed parallel programs. Our experimental measurements on a dual quad-core system demonstrate that OpenMP slightly outperforms TBB in our implementation.

TBB vs. OpenMP: What the official FAQ says

→ What about OpenMP?

Everyone should use OpenMP as much as they can. It is easy to use, it is standard, it is supported by all major compilers, and it exploits parallelism well. But it is very loop oriented, and does not address algorithm or data structure level parallelism.

To sum it up for MITK developers

- **TBB**: C++ library for parallel computing: Focuses on tasks not threads, nice programming style
- **OpenMP**: Same basic concept, but not C++, loop oriented
- **ITK, Qt Threads**: Abstraction layers to Windows Threads or Unix pthreads