Intel TBB
(Threading Building Blocks)

Rafael Asenjo, Mª Ángeles González
Dept. of Computer Architecture.
University of Málaga
http://www.ac.uma.es/~asenjo

Agenda

• Introduction.
  – Work Stealing. Installing and using
• Basic Classes
  – parallel_for, reduce, do, pipeline,…
• Advanced Classes
  – containers, locks, atomics, alloc, graphs,…
• Implementation details
  – Dealing with tasks. Examples
  – Recipes and advices
  – Research results
Introduction

• Yes, we are in the multicore era!
  – Regarding execution performance
    • Not a HW main problem any more
    • But a SW responsibility → apps parallelization
  – Sequential program → Slow program
  – SW developer → Parallelizer
  – Think in parallel

• Needed:
  – New languages and tools to help developers

Parallel languages and tools

• Distributed memory
  – Message passing: MPI, SHMEM, GASNet, …

• Shared memory
  – Pthreads, OpenMP
  – Task frameworks:
    • Intel TBB, MS TPL VS2010, Java Concurrency
    • Cilk, Intel CnC, OpenMP 3.0

• Parallel Languages
  • Partitioned Global Address Space (PGAS)
    – UPC, Co-array Fortran (CAF), Titanium (Parallel Java)
  • High Performance Computing Systems (HPCS)
    – Chapel (Cray), Fortres (Sun), X10 (IBM)
Task frameworks

- Tasks: much lighter weight than threads
  - Typically a function or object method
  - TBB: Task vs. thread
    - Linux: 18x faster to start and terminate a task
    - Windows: 100x faster
  - GCD (Apple’s Grand Central Dispatch)
    - Create a traditional thread: 100s instructions
    - Create a task: 15 instructions
  - In general: user level scheduler
    - The OS kernel do not see the tasks
    - The task scheduler mainly focus on performance

Work stealing scheduler

Thread Pool: Work-Stealing

Global Queue

Local Queue

Worker Thread 1

Task 1

Task 2

Program Thread

Task 4

Task 5

Core 1

Worker Thread p

Task 6

Core 2
Scheduler advantages

- **Automatic load balancing**
  - With less overhead than work-sharing

- **Without the OS scheduler restrictions**
  - Unfair scheduler (no preemption, no RR)
  - Sacrifices fairness for efficiency
  - It can be guided by the user
    - You don’t need to be “root” nor rely in RT priorities
    - Cache conscious
  - Avoid oversubscription, and therefore:
    - Avoid context switching overhead
    - Avoid cache cooling penalty
    - Avoid lock preemption and convoying
Task frameworks flavors (I)

- Cilk ('94, MIT)
  - Cilk Arts, Inc. (MIT spin-off) → Cilk++
  - Intel compra Cilk Arts (2009) -> Intel Cilk Plus
  - Basic constructors (linguistic extensions to C++)
    - cilk_spawn, cilk_sync, cilk_for
- Tasks in java.util.concurrent (Doug Lee, ‘99)
  - Later as part of JSR 166 (Java Spec. Request)
  - FJTaskRunner class: work-stealing scheduler
  - FJTask.fork() method: spawn a task
  - In 2011 Oracle’s Java 7: ForkJoinTask

Task frameworks flavors (II)

- OpenMP 3.0 (Task support, 2008)
  - task and taskwait constructs
  - gcc 4.3, icc/ifort 11, Nanos++, IBM XL C/C++
- Microsoft TPL in .NET 4.0 (2010)
  - Task Parallel Library component of the Parallel extensions
  - Main classes: Parallel.Invoke, Parallel.for
- Others:
  - Nanox RT (Alex Duran, BSC)
  - Qthreads Library (Sandia Natl. Lab.)
  - HPCS languages (X10-XWS, Chapel)
Threading Building Blocks

- Version 1.0 (August 29, 2006),
  - A year after Intel’s first dual-core x86 processor, the Pentium D
  - auto_partitioner: automatic grain size parameter.
- Version 2.0 (Jul. 2007). GPL version
  - Commercial version still available (same functionality as the open source version)
- Version 2.1 (Jul. 2008)
  - task-to-thread affinity, cancellation support, exception handling,…
  - Features support for lambda functions in C++0x
- Version 3.0 (May 2010).
  - Condition variables, design pattern manual, concurrent_unordered_map
- Version 4.0 (Sep. 2011).
  - Flow graphs, task and task group priorities, memory pools,…

TBB history

Languages
- Cilk
  - space efficient scheduler
  - cache-oblivious algorithms
- OpenMP*
  - fork/join tasks
- Threaded-C
  - continuation tasks
  - task stealing
- openMP taskqueue
  - while & recursion
- JSR-166 (FJTask) containers
- ECMA .NET*
  - parallel iteration classes
- Intel® TBB
- STAPL
  - recursive ranges
- STPL
  - small tasks
- STL
  - generic programming

*Other names and brands may be claimed as the property of others

Source: Arch D. Robison
Threading Building Blocks

• Advantages:
  – Task parallelism (faster than threads)
  – Load balance due to work-stealing
  – Generic programming (like STL)
  – Interoperability with other threading packages
  – Locality conscious to some extent
  – Productivity improvement
  – Open source: www.threadingbuildingblocks.org
    • Available for linux, windows, mac-os and sun

• Some limitations:
  – For C++
  – For shared memory

TBB constructors

• parallel_for
  – 1D, 2D, or custom iteration distrib. (block-cycl)
  – affinity_partitioner for cache
• parallel_reduce
• parallel_do
• pipeline
• containers
  – concurrent_hash_map
  – concurrent_vector
  – concurrent_queue
• spin_mutex
• atomic (atomic<int> var; //overload ++, --, -=.. operators)
• timing (tick_count)
• memory allocation (tbb_allocator<T>,cache_aligned_allocator<T>)
• task scheduling with task stealing
TBB classes

**Generic Parallel Algorithms**
- parallel_for
- parallel_reduce
- parallel_scan
- parallel_do
- pipeline
- parallel_sort
- parallel_invoke

**Concurrent Containers**
- concurrent_hash_map
- concurrent_queue
- concurrent_vector

**Memory Allocation**
- tbb_allocator
- cache_aligned_allocator
- scalable_allocator

**Synchronization Primitives**
- atomic, mutex, recursive_mutex
- spin_mutex, spin_rw_mutex
- queuing_mutex, queuing_rwlock_mutex
- null_mutex, null_rwlock_mutex

**Task scheduler**
- task_group
- task
- task_scheduler_init

**Threads**
- tbb_thread

Installing and using TBB

- **Download sources (_src.tgz file) from**
  - [www.threadingbuildingblocks.org](http://www.threadingbuildingblocks.org)
- **Unpack and “make info” to see the set up**
- **Do “make”**
  - In the build subdirectory there will be 2 new subdirectories (*_debug and *_release)
  - Use the _debug first
    - cd *_debug; source tbbvars.sh
    - In your makefiles use –ltbb_debug
  - When this work, use _release
    - cd *_release; source tbbvars.sh
    - In your makefiles use –ltbb
Agenda

✓ Introduction.
  – Work Stealing. Installing and using

 ➢ Basic Classes
  – parallel_for, reduce, do, pipeline,…

• Advanced Classes
  – containers, locks, atomics, alloc, graphs,…

• Implementation details
  – Dealing with tasks. Examples
  – Recipes and advices
  – Research results

---

Initializing the task scheduler

• TBB automatically initializes the scheduler
  – The first time a thread uses task scheduling services
  – Destroys it when the last such thread exits

• task_scheduler_init can be used to control:
  – When the task scheduler is constructed and destroyed.
  – The number of threads used by the task scheduler.
  – The stack size for worker threads.
Example: scaling study

```cpp
// Sketch of one way to do a scaling study
#include <iostream>
#include "tbb/task_scheduler_init.h"
int main() {
  int n = task_scheduler_init::default_num_threads();
  for( int p=1; p<=n; ++p ) {
    // Construct task scheduler with p threads
task_scheduler_init init(p);
    tick_count t0 = tick_count::now();
    //... execute parallel algorithm using task or
    //   template algorithm here...
tick_count t1 = tick_count::now();
    double t = (t1-t0).seconds();
    cout << "time = " << t << " with " << p << "threads\n";
  }
  return 0;
}
```

parallel_for (l)

- For parallel loops
- Example:

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

Will parallelize by partitioning the iteration space into chunks
# parallel_for (II)

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

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) {}
};

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

# parallel_for (III)

- **Notes:**
  - ApplyFoo(a) is a **body object** in which operator() processes a chunk
  - The argument to operator() is a template class: blocked_range<T>
    - The constructor is blocked_range<T>(begin,end,grain)
    - The iteration space is [begin,end)
  - This body object needs:
    - A copy constructor (invoked to create copies)
    - A destructor (to destroy the copies)
    - The implicitly generated methods may work
  - Method operator() is declared const
    - So it can not modify the variable members
How this works on blocked_range2d

Split range...

.. recursively...

.. until ≤ grainsize.

tasks available to thieves

Work Depth First; Steal Breadth First

Best choice for theft!
• big piece of work
• data far from victim’s hot data.

Second best choice.
Lambda Expressions

• For compilers supporting C++0x
  – It saves the coding time of creating the object

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

void ParallelApplyFoo( float* a, size_t n ) {
    parallel_for( blocked_range<size_t>(0,n),
                  [=](const blocked_range<size_t>& r) {
                      for(size_t i=r.begin(); i!=r.end(); ++i)
                          Foo(a[i]);
                  });
}
```

Lambda Expressions

• Even more compact:
  – Only for parallel_for over integers
  – Only 1D iteration space

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

void ParallelApplyFoo(float a[], size_t n) {
    parallel_for(size_t(0), n, [=](size_t i) {Foo(a[i]);});
}
```

– Compile with:
  • icc -std=c++0x foo.cpp (icc 11.1 or later)
  • g++ -std=c++0x foo.cpp (gcc 4.3 or later)
Controlling chunking

• By default, chunk size is automatic
  – heuristic: limit overhead while providing opportunities for load balancing
• To set it manually:
  – simple_partitioner() as the 3rd parallel_for arg.
  – grainsize in the blocked_range<T>(b,e,grainsize)

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

• simple_partitioner guarantees that: $G/2 \leq \text{chunksize} \leq G$
  – G can be also specified for auto_partitioner
• Then, it is guaranteed that: $G/2 \leq \text{chunksize}$
• Can be useful if the automatic heuristic fails

Grainsize selection

• Tradeoff between overhead and parallelism

  too fine $\Rightarrow$ too coarse $\Rightarrow$
  scheduling overhead dominates  lose potential parallelism
Grainsize selection

• Enough workload per chunk
  – For parallel_for, at least 100,000 ck per chunk
  – If an iteration takes 100ck, G should be 1000 at least.
• But enough chunks to feed the cores
  – If the loop has 2000 iterations and G=1000
    • Only two chunks to cores will be working
    • There is no work-stealing opportunities

Cache affinity

• Use affinity_partitioner when:
  – The computation does a few operations per data access.
  – The data acted upon by the loop fits in cache.
  – The loop, or a similar loop, is re-executed over the same data.
parallel_reduce (I)

- Serial code:

```c
float SerialSumFoo( float a[], size_t n ) {
    float sum = 0;
    for( size_t i=0; i!=n; ++i )
        sum += Foo(a[i]);
    return sum;
}
```

- Parallel code:

```c
float ParallelSumFoo( const float a[], size_t n ) {
    SumFoo sf(a);
    parallel_reduce( blocked_range<size_t>(0,n), sf );
    return sf.my_sum;
}
```

parallel_reduce (II)

- Body object:

```c
class SumFoo {
    float* my_a;
public:
    float my_sum;
    void operator()( const blocked_range<size_t>& r ) {
        float *a = my_a;
        float sum = my_sum;
        size_t end = r.end();
        for( size_t i=r.begin(); i!=end; ++i )
            sum += Foo(a[i]);
        my_sum = sum;
    }
    SumFoo( SumFoo& x, split ): my_a(x.my_a), my_sum(0) {}
    void join( const SumFoo& y ) {my_sum+=y.my_sum;}
    SumFoo(float a[] ): my_a(a), my_sum(0) {}  
};
```
parallel_reduce (III)

• Split-join sequence:

parallel_reduce (IV)

• The splitting constructor is not necessarily executed:
  – If there is no available worker:
• Caution:
  – Do not discard earlier acc.
  – Example:

```cpp
class SumFoo {
  ...
  public:
    float my_sum;
    void operator()( const blocked_range<size_t>& r ) {
      ...
      float sum = 0; // WRONG – should be "sum = my_sum"
      for( ... )
        sum += Foo(a[i]);
      my_sum = sum;
    }
  ...
};
```
More advanced topics

- Aside from the blocked_range<T>
  - blocked_range2d<T> also available
  - You can define your own iteration space object
    - Providing two boolean methods and a splitting constructor

```cpp
class my_own_range {
  // True if range is empty
  bool empty() const;
  // True if range can be split into non-empty subranges
  bool is_divisible() const;
  // Split r into subranges r and *this
  my_own_range( my_own_range& r, split );
...
};
```

- Samples
  - examples/parallel_for/tachyon uses blocked_range2d

parallel_do (I)

- The iteration space is not known in advance
  - Even, new iterations can be added on the fly

- Serial code:

```cpp
void SerialApplyFooToList( const std::list<Item>& list ) {
  for( std::list<Item>::const_iterator i=list.begin() ; i!=list.end() ; ++i )
    Foo(*i);
}
```

- Parallel code:

```cpp
class ApplyFoo {
public:
  void operator()( Item& item ) const {
    Foo(item);
  }
};

void ParallelApplyFooToList( const std::list<Item>& list ) {
  parallel_do( list.begin(), list.end(), ApplyFoo() );
}
```
parallel_do (II)

• Notes:
  – Each item is processed in parallel
    • It should take several thousand instructions
  – But items are fetched sequentially
    • Ensures that two thread will not take the same item
  – To enable parallel acquisition of work
    • If the iterators are random-access iterators
    • If in the body new items are added to the loop
      – Using feeder.add(item)

• Examples of feeder.add uses
  – Tree traversals: add descendant nodes after proc.
    • See examples/paralle_do/parallel_preorder
  – Wavefront computation: add dependent nodes

Pipeline

• Mimics the traditional assembly line
  – Data flows through a series of filter or stages
  – Each filter process the data in some way
  – Each filter can operate in serial or in parallel
  – TBB classes: pipeline and filter
  – Example:
    • Calculate the root mean square (RMS) of an array
      \[ x_{\text{rms}} = \sqrt{\frac{1}{n} \left( x_1^2 + x_2^2 + \cdots + x_n^2 \right)} \]
Pipeline main function

```c
int main()
{
    float array[10]={1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0};
task_scheduler_init init(NTHREAD);
pipeline mypipeline;

    MyInputFilter mif(array,array+10);
    mypipeline.add_filter(mif);

    MyProcessFilter mpf;
    mypipeline.add_filter(mpf);

    MyOutputFilter mof;
    mypipeline.add_filter(mof);

    // Run the pipeline
    mypipeline.run(NTOKENS);
    printf("RMS = %f\n", sqrt(mof.sum/mof.n));

    //Remove filters from pipeline before they are implicitly destroyed.
    mypipeline.clear();
}
```

Pipeline input filter

```c
/*----------STRUCTURE----------*/
typedef struct {float val; int idx;} item_t;

class MyInputFilter: public tbb::filter{
    float *first, *last;
    int index;

    public:
    MyInputFilter(float *first_, float *last_):
        filter(/*is_serial*/true),first(first_),last(last_),index(1){}
    
    void* operator()(void*){
        item_t * it;
        it = (item_t*) calloc(1,sizeof(item_t));

        if(first<last) {
            it->val=*first++;
            it->idx=index++;
            return (void*)it;
        } 
        else
            return NULL;
    }
};
```
Pipeline process filter

class MyProcessFilter: public tbb::filter{
  public:
    MyProcessFilter():filter(/*is_serial*/false){};

    void * operator()(void* item){
      item_t *it;
      it=(item_t*) item;
      it->val=it->val * it->val;
      return (void*) it;
    }
};

  – Receives a void* item from the input
  – Cast to item_t*
  – Computes the squares of the value
  – Send the updated item to the next filter (after casting)

Pipeline output filter

class MyOutputFilter: public tbb::filter{
  public:
    float sum;
    int n;
    MyOutputFilter():filter(/*is_serial*/true),sum(0) {};

    void * operator()(void* item){
      item_t *it;
      it=(item_t*) item;
      printf("Received item with value: %f, index: %d\n",it->val,it->idx);
      sum+=it->val;
      n=it->idx;
      free(it);
    }
};

  – Receives a void* item from the process filter, cast,…
  – Accumulates in sum and saves the last index
  – Frees the memory used by the item
  – MyOutputFilter.sum and MyOutputFilter.n are public
Pipeline using lambda

```cpp
float RootMeanSquare(float* first, float* last) {
    float sum = 0;
    parallel_pipeline(/*ntokens=*/16,
        make_filter<void,float*>(
            filter::serial,
            [&](flow_control& fc)-> float*
            { if( first<last ) {
                return first++;
            } else {
                fc.stop();
                return NULL;
            }
            } &
        ) &
        make_filter<float*,float*>(
            filter::parallel,
            [](float* p){return (*p)*(*p);}
        ) &
        make_filter<float,void*>(
            filter::serial,
            [&](float x) {sum+=x;}
        )
    );
    return sqrt(sum);
}
```

Pipeline: intermediate filter

```cpp
void* MyTransformFilter::operator()(void* item){
    struct in_data * input;
    input= (struct in_data *) item;

    struct out_data * output;
    output=(struct out_data*)calloc(...);

    // Process input
    // Create output

    free(input);
    return (void *) output;
}
```
Parallel pipeline

Serial stage processes items one at a time in order.

Items wait for turn in serial stage

Parallel stage scales because it can process items in parallel or out of order.

Another serial stage.

Throughput limited by throughput of slowest stage.

Controls excessive parallelism by limiting total number of items flowing through pipeline.

TBB pipeline details

- max # tokens: max # of tasks in-flight →
  - max # of items being processed in the pipe
  - each token is created for each input item
    - the token travels through the pipeline and die
  - A task is spawned for each token
    - The task lives in a working thread (1 per core)
    - Efficient use of the cache
      - each worker thread takes an item through as many stages as possible,
      - the algorithm is biased toward finishing old items before tackling new ones

Source: Arch D. Robison

27/01/12 TBB tutorial. Rafael Asenjo. 45
Pipeline notes

- Sub-problem size that each token process
  - Large enough to avoid overhead
  - But small enough to fit in cache
- Only linear pipelines allowed:

- This alternative affect latency, but not throughput
- A, B, D and E filters need to be modified to properly handle objects that shouldn’t be there.

Other basic classes (I)

- Parallel prefix or scan: parallel_scan
  - Takes
    - a binary associative operator, for instance +
    - an array of elements, for instance [3 1 7 0 4 1 6 3]
  - Returns the running sum: [3 4 11 14 16 22 25]
  - Used in: sorting, lexical analysis, string comp, …
  - Example

```c
float DoParallelScan( T y[], const T x[], int n ) { 
    Body body(y,x);
    parallel_scan( blocked_range<int>(0,n), body );
    return body.get_sum();
}
```

- (see the Body class in Reference man. page 45)
Other basic classes (II)

- **Parallel sort**
  - Sort the sequence [begin,end) (rand. access iterators)
  - A compare method should be available
    - By default std::less<T> is used
    - So parallel_sort(i,j) ≈ parallel_sort(i,j,std::less<T>)

```cpp
float a[N];
float b[N];
void SortExample() {
    for (int i = 0; i < N; i++) {
        a[i] = sin((double)i);
        b[i] = cos((double)i);
    }
    parallel_sort(a, a + N);
    parallel_sort(b, b + N, std::greater<float()?>);
}
```

Other basic classes (III)

- **Parallel_invoke**
  - parallel_invoke(f0, f1, f2, f3, ..., f9) (from 2 to 10 args)
  - Evaluates f0(), f1(), ... possibly in parallel
  - Each object argument must have operator() defined
  - Or can be pointers to functions:

```cpp
#include "tbb/parallel_invoke.h"
void f();
extern void bar(int);
class MyFunctor {
    int arg;
public:
    MyFunctor(int a) : arg(a) {} 
    void operator()() const {bar(arg);}
};
void RunFunctionsInParallel() {
    MyFunctor h(3);
    tbb::parallel_invoke(f, h);
}
```

Using lambda expressions

```cpp
#include "tbb/parallel_invoke.h"
void f();
extern void bar(int);
void RunFunctionsInParallel() {
    tbb::parallel_invoke( 
        f,
        []{bar(2);},
        []{bar(3);} 
    );
}
```
Agenda

✓ Introduction.
  – Work Stealing. Installing and using
✓ Basic Classes
  – parallel_for, reduce, do, pipeline,…
➢ Advanced Classes
  – containers, locks, atomics, alloc, graphs,…
• Implementation details
  – Dealing with tasks. Examples
  – Recipes and advices
  – Research results

Containers

• Concurrent containers
  – Multiple threads can access and update items
  – Parallel version of the C++ STL containers
    • Used in parallel by locking the whole container \( \rightarrow\) concurr. 🔽
  – In contrast, TBB containers provide concurrency by:
    • Fine-grain locking: only the regions that really need the lock
    • Lock-free techniques: in some cases locks are not needed
  – Flavors:
    • concurrent_hash_map
    • concurrent_vector
    • concurrent_queue
    • concurrent_unordered_map
    • concurrent_unordered_set
concurrent_hash_map (I)

- Example:
  - Counter the number of occurrences (value) of the strings (key)

```
<table>
<thead>
<tr>
<th>keys</th>
<th>hash function</th>
<th>buckets</th>
</tr>
</thead>
<tbody>
<tr>
<td>hello</td>
<td></td>
<td>000</td>
</tr>
<tr>
<td>house</td>
<td>001</td>
<td>house</td>
</tr>
<tr>
<td></td>
<td>002</td>
<td></td>
</tr>
<tr>
<td>dog</td>
<td></td>
<td>151</td>
</tr>
<tr>
<td></td>
<td>152</td>
<td>hello</td>
</tr>
<tr>
<td></td>
<td>153</td>
<td>chair</td>
</tr>
<tr>
<td></td>
<td>154</td>
<td></td>
</tr>
<tr>
<td>table</td>
<td></td>
<td>253</td>
</tr>
<tr>
<td></td>
<td>254</td>
<td>dog</td>
</tr>
<tr>
<td></td>
<td>255</td>
<td></td>
</tr>
<tr>
<td>overflow entries</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>table</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
</tr>
</tbody>
</table>
```

```
// A concurrent hash table that maps strings to ints.
typedef concurrent_hash_map<string, int, MyHashCompare> StringTable;

const size_t N = 1000000;
string Data[N];

void CountOccurrences(int i) {
    // Construct empty table.
    StringTable table;
    // Put occurrences into the table
    parallel_for( blocked_range<string*>( Data, Data+i, 1000 ),
                  Tally(table));
    // Display the occurrences
    for( StringTable::iterator i=table.begin(); i!=table.end(); ++i )
        printf("%s %d\n", i->first.c_str(), i->second);
}
```
concurrent_hash_map (III)

- Third parameter of concurrent_hash_map
  - struct MyHashCompare. Provides:
    - hash function
    - key comparison function

```cpp
// Structure that defines hashing and comparison operations
struct MyHashCompare {
    static size_t hash( const string& x ) {
        size_t h = 0;
        for( const char* s = x.c_str(); *s; ++s )
            h = (h*17)^*s;
        return h;
    } //! True if strings are equal
    static bool equal( const string& x, const string& y ) {
        return x==y;
    }
};
```

concurrent_hash_map (IV)

- Body object for the parallel_for
  - Difference between accessor and const_accessor

```cpp
// Function object for counting occurrences of strings.
struct Tally {
    StringTable& table;
    Tally( StringTable& table_ ) : table(table_) {};
    void operator()( const blocked_range<string*> range ) const {
        StringTable::accessor a;
        table.insert( a, *p );
        a->second += 1;
    }
};
```

```cpp
StringTable::const_accessor ac;
const string a="house";
table.find(ac,a);
cout << ac->first.c_str() << " has " << ac->second << " occurrences" << endl;
```
concurrent_hash_map (V)

- The accessors are fine-locking the hash table
  - Release them as soon as possible
  - Automatically released at the end of the block
  - Can be manually released by using method release()
    - Useful if you can release it sooner than the end of the block

```cpp
StringTable accessor a; //declared outside the for block
for( string* p=range.begin(); p!=range.end(); ++p ) {
    table.insert( a, *p );
    a->second += 1;
    a.release();
}
```

concurrent_vector (I)

- A dynamically growable array of T
  - concurrent_vector<T> main methods:
    - push_back(x) safely append x to the vector
    - grow_by (n) safely append n consecutive elements initialized with T()
    - grow_to_at_least (n) grows to size n if it is shorter
    - size() returns the number items (including pending items)
  - Example:
    - Use std::copy instead of strcpy (items might not be consec.)
    - grow_by returns an iterator to the first appended element

```cpp
void Append( concurrent_vector<char>& vector, char* string ) {
    size_t n = strlen(string)+1;
    std::copy( string, string+n, vector.grow_by(n) );
}
```
concurrent_vector (II)

- **Advantages:**
  - Safe to use the iterators while vector is growing
    - as long as the iterator never goes past the end() value
  - Elements are not moved until the array is cleared
    - better exploit cache

- **Disadvantages**
  - More overhead than std::vector
    - Use it when you need dynamic resize with in-flight accesses
  - Insertion and access need to be synchronized
  - clear() or destroying a vector are not concurrency safe.

concurrent_queue

- **concurrent_queue<T>**
  - push(T) just push an item onto back of the queue
  - try_pop(T) atomic operation → thread-safe
  - Not guaranteed to be FIFO in concurrent scenario
    - “first” is uncertain if there are threads pushing and popping
    - However, if one thread pushes two values and another one pops them, they will be popped in the same order
  - Unbounded size and no synchronization provided

```cpp
struct item { int i, j;};
concurrent_queue<item> *TBBqueue;
int main(){
    TBBqueue = new concurrent_queue<item>();
    item A={0,1};  TBBqueue->push(A);
    item B={3,9};  TBBqueue->push(B);
    item C;
    while (TBBqueue->try_pop(C)) //while there are entries...
        cout << C.i << " , " << C.j << endl;
}
```
**concurrent_bounded_queue**

- **concurrent_bounded_queue<T>**
  - Capacity can be specified
    - Using `set_capacity(size_t cap)`
    - Otherwise unbounded (until memory runs out)
  - Adds blocking operations
    - `pop(T)` waits until it can succeed
    - `push(T)` waits until it can succeed without overflowing
    - `try_push(T)` pushes only if it does not exceed queue capacity
    - `size()` returns a signed integer
      - `# push - # pops \to size() = -n` if there are `n` pending pop operations.
      - `empty() = true` if `size() <= 0`
  - Bounded queues are slower than unbounded ones

**queues are not advisable**

- Typically used in producer-consumer problems
- The downside:
  - A queue is a bottleneck (FIFO order constraint)
  - A thread popping may have to wait idly
  - Not cache conscious:
    - While waiting for the pop, the pushed value becomes “cold”
    - Or the thread popping is different and the item must be moved
- Alternative: `parallel_do` and `pipeline`
  - Optimized to do other work until a value shows up
  - Cache conscious
    - A thread adds an item that is kept local to that thread
    - Unless stealing, the “hot” thread will process it
Mutual Exclusion

- To control the threads that can run a code region simultaneously
- Implemented by mutex and locks
  - Designed for high performance
    - Does not require the programmer to release the lock
    - Releases the lock if an exception is thrown while in the crit. sec.
  - All mutexes have the same interface
    - For example, all of them have scoped_lock(mutex)
  - Types:
    - spin_mutex, spin_rw_mutex
    - queuing_mutex, queuing_rw_mutex
    - null_mutex, null_rw_mutex
    - mutex, recursive_mutex

```cpp
#include <iostream>
#include "tbb/spin_mutex.h"
#include "tbb/parallel_invoke.h"
using namespace std;
using namespace tbb;

typedef spin_mutex MyMutex_t;
MyMutex_t mutex;
unsigned long p=0;

void doit(){
    unsigned long i,j,t;
    for (i=0; i<10; i++){
        MyMutex_t::scoped_lock mylock(mutex);
        for (j=0; j<1000000; j++) p++;
        t=p;
        cout << "Value of t: " << t << endl;
    }
}

int main () {
    tbb::parallel_invoke(doit, doit);
    cout << "Program end" << endl; }
```

**spin_mutex**

Appropriate when lock is quickly released

- Use typedef so you can change the mutex type easily
- Do not forget the lock object name (`mylock`) (otherwise it is released at the `;``

- mylock is automatically released here
spin_mutex

- Release the lock as soon as possible
  - lock.acquire waits on the lock until it can acquire it

```c
void doit() {
    unsigned long i, j, t;
    for (i=0; i<10; i++) {
        MyMutex_t::scoped_lock lock;
        lock.acquire(mutex);
        for (j=0; j<1000000; j++) p++;
        t=p;
        lock.release();
        cout << "Value of t: " << t << endl;
    }
}
```

- The use of extra braces {} is advisable to clarify to maintainers which code is protected by the lock

Mutex properties

- Scalable:
  - Non scalable mutex consume processor ck and memory bandwidth
  - However, scalable mutex are often slower (under light contention)
- Fair:
  - Lets threads through in the order they arrive
  - However unfair threads may be faster: lets threads running go through first
- Recursive:
  - A thread that is already holding a lock can acquired another lock on the same mutex (for recursive prog.)
- Yield or Block
  - The thread that is waiting can yield or block
Mutex flavors

<table>
<thead>
<tr>
<th>Mutex</th>
<th>Scalable</th>
<th>Fair</th>
<th>Rec.</th>
<th>Long Wait</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>mutex</td>
<td>OS dep.</td>
<td>OS dep.</td>
<td>no</td>
<td>blocks</td>
<td>≥3 words</td>
</tr>
<tr>
<td>recursive_mutex</td>
<td>OS dep.</td>
<td>OS dep.</td>
<td>✓</td>
<td>blocks</td>
<td>≥3 words</td>
</tr>
<tr>
<td>spin_mutex</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yields</td>
<td>1 byte</td>
</tr>
<tr>
<td>queuing_mutex</td>
<td>✓</td>
<td>✓</td>
<td>no</td>
<td>yields</td>
<td>1 word</td>
</tr>
<tr>
<td>spin_rw_mutex</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yields</td>
<td>1 word</td>
</tr>
<tr>
<td>queuing_rw_mutex</td>
<td>✓</td>
<td>✓</td>
<td>no</td>
<td>yields</td>
<td>1 word</td>
</tr>
<tr>
<td>null_mutex</td>
<td>moot</td>
<td>✓</td>
<td>✓</td>
<td>never</td>
<td>empty</td>
</tr>
<tr>
<td>null_rw_mutex</td>
<td>moot</td>
<td>✓</td>
<td>✓</td>
<td>never</td>
<td>empty</td>
</tr>
</tbody>
</table>

- **Use spin_mutex or rethink your code**
  - If the mutex is heavily contended, scalability is lost

Mutex flavors

- **mutex and recursive_mutex** are OS wrappers
  - Windows: on top of CRITICAL_SECTION
  - Linux and MacOS: on top of Pthreads mutex
  - Advantages: portability and exception-safe interface

- **null_mutex**: do nothing
  - Useful when the mutex type is a parameter

- **Read mutex ( _rw_ versions)**
  - Allow multiple readers to go through
  - For readers: lock(mutex,false)
  - For writers: lock(mutex,true) = lock(mutex)
Mutex pathologies

• Avoid deadlock:
  – Avoid needing to hold two locks
  – Acquire locks in the same order
  – Do not use locks → better rely in atomic vars.

• Convoying:
  – When the OS interrupts a thread holding a lock
  – All other threads waiting for that lock are idle until the
    thread resumes and release the lock
  – With fair locks is even worse: if a waiting thread is not
    ready, the other threads in the queue have to wait
  – Release the lock ASAP and use atomic instead.

Atomic operations

• Advantages
  – You can get rid of mutual exclusion in many cases
  – Atomic ops are seen as instantaneous by other threads
  – Relatively quicker than locks

• Limitations:
  – Only few operations available
    • ++, --, -=, and +=
    • x.fetch_and_store(y): x=y and return old value of x
    • x.fetch_and_add(y): x+=y and return old value of x
    • x.compare_and_swap(y,z): if(x==z) x=y; return old x
    • Atomic capture supported: if(x-- == 0) action() is safe
  – atomic<T> where T is integral, enum or pointer type.
Atomic operations

- **compare_and_swap example**
  - How to atomically update a shared variable as a function of the old value of that variable: $x = f(x)$

```cpp
atomic<int> globalx;
int UpdateX() { // Update x and return old value of x.
    do {
        // Read globalX
        oldx = globalx;
        // Compute new value
        newx = ...expression involving oldx.... f(oldx)
        // Store new value if another thread has not changed globalX.
        } while( globalx.compare_and_swap(newx,oldx)!=oldx );
    return oldx;
}
```

- If function $f$ takes only a few instructions this idiom is faster than the corresponding mutex solution

---

Atomic operations

- **atomic<T> has no constructor**

- As for any C++ class with no declared constructor
  - Initialized to zero when:
    - $x$ is declared as a file-scope variable or a static data member
    - $x$ is a member of a class and it is listed in the constructor list

```cpp
atomic<int> x; // zero-initialized because it is at file scope
class Foo {
    atomic<int> y;
    atomic<int> notzeroed;
    static atomic<int> z;
public:
    Foo() : y() // zero-initializes y.
    { // notzeroed has unspecified value here.
    }
};
atomic<int> Foo::z; // zero-initialized because it is a static member
```
Memory Allocation

• Scalability: scalable_allocator<T>
  – Enables concurrent allocation operations
  – Useful when threads frequently allocates and free mem.

• False sharing: cache_aligned_allocator<T>
  – When different vars are allocated in the same line
  – The line has to be moved among cores (100’s ms)
  – This allocator use a single line for each var
    • Two vars allocated like that are in different lines
    • However, if one var is std::allocator-ed this doesn’t hold

• Can be used for containers:
  – std::vector<int, cache_aligned_allocator<int>>;

Timing

• Use class tick_count:

```
tick_count t0 = tick_count::now();
... do some work ...
tick_count t1 = tick_count::now();
printf("work took %g seconds\n", (t1-t0).seconds());
```

• Advantages:
  – Thread safe
    • Valid to subtract tick_count values created by different threads
  – Easy conversion to seconds
  – Based on the wall clock time
  – Bases on the highest time resolution available on the platform
Flow Graph

- There are some applications that are best organized as computations that explicitly pass messages.
- These messages may contain data or simply act as signals that a computation has completed.
- TBB 4 introduce a new class GRAPH.
  - This class and its associated nodes can be used to express such applications.
- There are three types of components used to implement a graph: a graph object, nodes and edges.

Flow Graph example

- One graph with 5 nodes and 5 edges
- The input is a set of integers \( x=1..10 \)
  - The output is \( \text{sum } += x_i^2 + x_i^3 \)
Flow Graph example

```cpp
#include "tbb/flow_graph.h"
#include <cstdio>

using namespace std;
using namespace tbb;
using namespace tbb::flow;

struct square {
    int operator(int v) { return v*v; }
};

struct cube {
    int operator(int v) { return v*v*v; }
};

class sum {
    int &my_sum;
public:
    sum(int &s) : my_sum(s) {}
    int operator((std::tuple<int, int> v) {
        my_sum += std::get<0>(v) + std::get<1>(v);
        return my_sum;
    });
};

int main() {
    int result = 0;
    graph g;
    broadcast_node<int> input;
    function_node<int, int> squarer( g, unlimited, square() );
    function_node<int, int> cuber( g, unlimited, cube() );
    join_node< std::tuple<int, int>, queueing > join( g );
    function_node< std::tuple<int, int>, int > summer( g, serial, sum(result) );
    make_edge( input, squarer );
    make_edge( input, cuber );
    make_edge( squarer, std::get<0>( join.inputs() ) );
    make_edge( cuber, std::get<1>( join.inputs() ) );
    make_edge( join, summer );
    for (int i = 1; i <= 10; ++i) input.try_put(i);
    g.wait_for_all();
    printf("Final result is %d\n", result);
    return 0;
}
```
Graph

• The graph object is the owner of the tasks created on behalf of the flow graph.
• Users can wait on the graph if they need to wait for the completion of all of the tasks related to the flow graph execution.

Nodes

• Nodes invoke user-provided function objects or manage messages as the flow to/from other nodes.
• There are pre-defined nodes that buffer, filter, broadcast or order items as they flow through the graph.

Edges

• Edges are the connections between the nodes, created by calls to the make_edge function.
• Edges are classified in two sets:
  – Push set (S)
  – Pull set (L)
• The hybrid push-pull protocol used by tbb::graph biases communication to prevent polling and to reduce unnecessary retries.
• For each edge (Vs, Vr), Vs is the predecessor / sender and Vr is the successor / receiver:
  – When in the push set S: messages are initiated by the sender which tries to put to the receiver.
  – When in the pull set L: messages are initiated by the receiver, which tries to get from the sender
Body Objects

• Some nodes execute user-provided body objects.
• These objects can be created by instantiating function objects or lambda expressions.
• The nodes that use body objects include
  – continue_node
    • It executes a specified body object when triggered and broadcasts the generated value to all of its successors.
  – function_node
    • Receives messages at a single input port and generates a single output message that is broadcast to all successors.
  – join_node
  – source_node

Message passing protocol (MPP)

• A thread is not assigned to each tbb::graph node.
• Tasks are created on-demand to execute node bodies and pass messages between nodes when there is activity in the graph.
• Consequently, a tbb::graph node does not spin in a loop waiting for messages to arrive.
• Instead when a message arrives, a task is created to apply the receiving node’s body to the incoming message.
• Some nodes can reject an incoming message
Message passing protocol (II)

- Protocol for dealing with message rejection that is efficient and ensures that messages aren’t accidentally dropped.
- It’s important to not create many small tasks or tasks that waste resources by spinning.
- Creating a new task to retry at each rejection may generate many small useless tasks.
- Instead, tbb::graph uses a hybrid push-pull protocol as a more efficient alternative.

Message passing protocol (III)

- Edges dynamically switch between a push and pull protocol at rejections.
- If a message attempt across an edge fails, the edge is moved to the other set. For example, if a put across the edge (Vi, Vj) is rejected, the edge is removed from the push set S and placed in the pull set L.
Message passing protocol (IV)

- This protocol results in a reduction in the messages across an edge, while maintaining quick response times.
- If a sender, $V_i$, generates data at a faster rate than its successor $V_j$, the edge will transition into pull mode, eliminating the many rejections that $V_i$ would see if it were to continue to send.
- Likewise, if a receiver $V_j$ processes data faster than its sender $V_i$, the edge will stay in push mode, allowing $V_i$ to send data as soon as it is generated.

Exceptions

- When code inside TBB block throws exception:
  - Exception captured. Any further exception ignored
  - Algorithm cancelled. Pending iterat. not executed
  - Exception throws on the invoking thread

```cpp
vector<int> Data;
struct Update {
    void operator()( const blocked_range<int>& r ) const {
        for( int i=r.begin(); i!=r.end(); ++i )
            Data.at(i) += 1;
    }
};
int main() {
    Data.resize(1000);
    try {
        parallel_for( blocked_range<int>(0, 2000), Update());
    } catch( captured_exception& ex ) {
        cout << "captured_exception: " << ex.what() << endl;
    } catch( out_of_range& ex ) {
        cout << "out_of_range: " << ex.what() << endl;
    } return 0;
}
Cancellation

• Cancelling without throwing an exception:

```cpp
vector<int> Data;
struct Update {
    void operator()( const blocked_range<int>& r ) const {
        for( int i=r.begin(); i!=r.end(); ++i )
            if( i<Data.size() ) {
                ++Data[i];
            } else {
                // Cancel related tasks.
                if( task::self().cancel_group_execution() )
                    cout << "Index " << i << " caused cancellation\n";
                return;
            }
    }
};
int main() {
    Data.resize(1000);
    parallel_for( blocked_range<int>(0, 2000), Update() );
    return 0;
}
```

Agenda

✓ Introduction.
    – Work Stealing. Installing and using

✓ Basic Classes
    – parallel_for, reduce, do, pipeline,…

✓ Advanced Classes
    – containers, locks, atomics, alloc, graphs,…

➢ Implementation details
    – Dealing with tasks. Examples
    – Recipes and advices
    – Research results
Dealing with tasks

• The task scheduler powers the templates
  – If possible, use the templates
  – Sometimes there is no template fitting the requirement

• Classes and methods
  – task_scheduler_init Class
    • initialize(max_threads=automatic), terminate()
  – task Class
    • new(allocation_*)
    • spawn(task& t)
    • spawn_and_wait_for_all(task& t)
    • recycle_as_continuation()
    • recycle_as_child_of(task& new_successor)

Example: Fibonacci

• The Fibonacci numbers:
  \[ \langle 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots \rangle, \]
  Each number is the sum of the previous two.

```cpp
int fib(int n)
{
    if (n < 2) return n;
    else {
        int x = fib(n-1);
        int y = fib(n-2);
        return x + y;
    }
}
```

Recurrence:
\[ F_0=0, \]
\[ F_1=1, \]
\[ F_n = F_{n-1} + F_{n-2} \text{ for } n>1 \]
Pthread Implementation

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int fib(int n)
{
  if (n < 2) return n;
  else {
    int x = fib(n-1);
    int y = fib(n-2);
    return x + y;
  }
}

typedef struct {
  int input;
  int output;
} thread_args;

void *thread_func ( void *ptr )
{
  int i = ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  return NULL;
}

int main(int argc, char *argv[])
{
  pthread_t thread;
  thread_args args;
  int status;
  int result;
  int thread_result;
  if (argc < 2) return 1;
  int n = atoi(argv[1]);
  if (n < 30) result = fib(n);
  else {
    args.input = n-1;
    status = pthread_create(&thread, NULL,
                            thread_func, (void*) &args);
    // main can continue executing
    result = fib(n-2);
    // Wait for the thread to terminate.
    pthread_join(thread, NULL);
    result += args.output;
  }
  printf("Fibonacci of %d is %d.\n", n, result);
  return 0;
}
```

Issues with Pthreads

- **Overhead**: The cost of creating a thread >104 cycles ⇒ coarse-grained concurrency. (Thread pools can help.)
- **Scalability**: Fibonacci code gets about 1.5 speedup for 2 cores. It needs a rewrite for more cores.
- **Modularity and Code Simplicity**: The Fibonacci logic is no longer neatly encapsulated in the `fib()` function. Programmers must marshal arguments.
- **Efficiency**: Load-unbalance. Sometimes solved by oversubscription (OS responsibility). Bad idea:
  - Time of context switching
  - Cache cooling
  - Lock preemption
OpenMP Implementation

• Features
  • Using #pragma
  • Explicit sharing of memory
  • Task and taskwait directives

• More elegant than TBB

Syntax:

```cpp
int fib(int n) {
    if (n < 2) return n;
    int x, y;
    #pragma omp task shared(x)
    x = fib(n - 1);
    #pragma omp task shared(y)
    y = fib(n - 2);
    #pragma omp taskwait
    return x+y;
}
```

Cilk++ Implementation

• Cilk++
  • Small set of linguistic extensions to C++

• Constructors:
  • cilk_spawn: grant permission for parallel execution
  • cilk_sync: wait for children
  • cilk_for: parallel loop

Syntax:

```cpp
cilk int fib (int n) {
    if (n < 2) return n;
    int x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return x+y;
}
```
TBB Implementation

#include <iostream>
#include "tbb/task.h"

using namespace std;
using namespace tbb;

long SerialFib( long n ) {
    if ( n<2 )
        return n;
    else
        return SerialFib(n-1)+SerialFib(n-2);
}

long ParallelFib( long n ) {
    long sum;
    FibTask& a = *new(task::allocate_root()) FibTask(n,&sum);
    task::spawn_root_and_wait(a);
    return sum;
}

TBB Implementation

class FibTask: public task {
public:
    const long n;
    long* const sum;
    FibTask( long n_, long* sum_ ) : n(n_), sum(sum_) {}
    task* execute() { // Overrides virtual function task::execute
        if ( n<5 ) {
            *sum = SerialFib(n);
        } else {
            long x, y;
            FibTask& a = *new( allocate_child() ) FibTask(n-1,&x);
            FibTask& b = *new( allocate_child() ) FibTask(n-2,&y);
            // Set ref_count to "two children plus one for the wait".
            set_ref_count(3);
            // Start b running.
            spawn( b );
            // Start a running and wait for all children (a and b).
            spawn_and_wait_for_all(a);
            // Do the sum
            *sum = x+y;
        }
        return NULL;
    }
};
TBB Implementation

<table>
<thead>
<tr>
<th>Stack</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>main sum</td>
<td></td>
</tr>
<tr>
<td>Fib(3) x = 3</td>
<td></td>
</tr>
<tr>
<td>Fib(3) y = 1</td>
<td></td>
</tr>
<tr>
<td>Fib(2) x = 1</td>
<td></td>
</tr>
<tr>
<td>Fib(2) y = 0</td>
<td></td>
</tr>
</tbody>
</table>

new root FibTask(3, &sum)

=4

new child FibTask(2, &x)

new child FibTask(1, &y)

new child FibTask(1, &x)

new child FibTask(0, &y)

TBB task internals

- Example: Fibonacci
  - Each working thread has a ready pool of tasks

LIFO (depth-first) work
  - when a working thread ends his task takes another one from the deepest part

FIFO (breath-first) theft
  - if a working thread does not have any task in his pool, it steals from the shallowest side of ANOTHER thread pool

Task deque in a thread (double ended queue)

top (oldest task)

bottom (youngest task)
Recursive parallelism

- Each child task creates two child tasks.
- The same idea is exploited in `paralle_for`.

```
wait_for_all(task *child) {
   task = child;
   Loop until root is alive
   {
      do {
         while (task available) {
            next_task = task->execute();
            Decrease ref_count for parent of task;
            if (!next_task && ref_count==0)
               next_task = parent of task;
            task = next_task;
         }
      }
      task = get_task();
      while (task);
      task = steal_task(random());
      if (steal unsuccessful)
         Wait for a fixed amount of time;
      if (waited for too long)
         wait for master thread to produce new work;
   }
}
```

ref_count is not decremented if the task has call `recycle_as_continuation`
TBB scheduler

• More precisely:
  – After completing a task t, a thread chooses its next task according to the first applicable rule below:
    1. The task returned by t.execute()
    2. The successor of t if t was the last predecessor (refcount=0)
    3. A task from the end of the threads’ own deque
    4. A task with affinity for the thread
    5. A task popped from approximately the beginning of the shared queue
    6. A task popped from the beginning of another randomly chosen thread’s deque

All threads are in wait_for_all

// Start task scheduler
tbb::task_scheduler_init init( NTHREAD );

```cpp
Th0
  create_one_worker()
  (pthread_create)
  Th1
    worker_routine()
    create_worker()
    worker_routine()
  wait_for_all()

Th2
  worker_routine()
  create_worker()
  worker_routine()
  wait_for_all()

Th3
  worker_routine()
  create_worker()
  worker_routine()
  wait_for_all()
```
struct FibTask: public task {
    const long n;
    long* const sum;
    FibTask( long n_, long* sum_ ) : n(n_), sum(sum_) {}
    task* execute() {
        if( n<CutOff ){
            *sum = SerialFib(n);
            return NULL;
        } else {
            FibContinuation& C =
                *new( allocate_continuation() ) FibContinuation(sum);
            FibTask& a = *new( c.allocate_child() ) FibTask(n-1,&C.x);
            FibTask& b = *new( c.allocate_child() ) FibTask(n-2,&C.y);
            // Set ref_count to "two children plus one for the wait".
            C.set_ref_count(2);
            spawn( b );
            spawn( a );
            return NULL;
        }
    }
};

struct FibContinuation: public task {
    long* const sum;
    long x, y;
    FibContinuation( long* sum_ ) : sum(sum_) {}
    task* execute() {
        *sum = x+y;
        return NULL;
    }
};

Task continuation pasing

```
new root FibTask(3,&sum)  new cont
   3  &sum  &sum  x  y

new child Fib(2,&x)
   2  &sum  x  y

new child Fib(1,&y)
   1  &sum

new child FibTask(1,&x)
   1  &sum

new child FibTask(0,&y)
   0  &sum
```

Th0 deque

fibTask(1)

FibTask(0)

fibTask(1)

fibTask(1)

fibTask(1)
Scheduler bypass

```cpp
struct FibTask: public task {
    const long n;
    long* const sum;
    FibTask( long n_, long* sum_ ) : n(n_), sum(sum_) {}  
    task* execute() {
        if( n<CutOff ) {
            *sum = SerialFib(n);
            return NULL;
        } else {
            FibContinuation& C =
                *new( allocate_continuation() ) FibContinuation(sum);
            FibTask& a = *new( c.allocate_child() ) FibTask(n-1,&c.x);
            FibTask& b = *new( c.allocate_child() ) FibTask(n-2,&c.y);
            // Set ref_count to "two children plus one for the wait".
            c.set_ref_count(2);
            spawn( b );
            spawn( a );
            return &a;
        }
    }
};
```

Recycling

```cpp
struct FibTask: public task {
    const long n;
    long* const sum;
    FibTask( long n_, long* sum_ ) : n(n_), sum(sum_) {}  
    task* execute() {
        if( n<CutOff ) {
            *sum = SerialFib(n);
            return NULL;
        } else {
            FibContinuation& C =
                *new( allocate_continuation() ) FibContinuation(sum);
            FibTask& a = *new( c.allocate_child() ) FibTask(n-1,&c.x);
            FibTask& b = *new( c.allocate_child() ) FibTask(n-2,&c.y);
            recycle_as_child_of( c );
            n-=1;
            sum = &c.x;
            // Set ref_count to "two children plus one for the wait".
            c.set_ref_count(2);
            spawn( b );
            return &a;
        }
    }
};
```
Recycling

new root FibTask(3,&sum)  new cont
2  &sum  &sum x^4 y^1
1  &sum  &sum x^1 y^0

recycle as child
new child Fib(1,&y)
1  &sum

recycle as child
new child FibTask(0,&y)
0  &sum

Inside pipeline.cpp

// Short task scheduler
TBB: :task_scheduler_init init( NUM_THREADS );

// Create the pipeline
tbb: :pipeline pipeline;

// Create input filter and add it to the pipeline
MyInputFilter input_filter; pipeline.add_filter( input_filter );

// Create SEG stage and add it to the pipeline
MySegF seg_filter; pipeline.add_filter( seg_filter );

// Create OUT stage and add it to the pipeline
MyOutF out_filter; pipeline.add_filter( out_filter );

// Run the pipeline
TBB: :tick count c0 = tbb: :tick_count::now();
TBB: :tick count c1 = tbb: :tick_count::now();

• A new pipeline object
  – filter_list (pointer to the list with the stages)
  – end_counter (the task that signals the work is done)
  – token_counter (# of tokens created so far)
  – input_tokens (# of idle tokens waiting for input)

• Create each filter object
• Append each filter to the pipeline’s filter_list

ordered buffers for the serial filters

• Run the pipeline with max n_tokens
  – allocates end_counter task (the root task)
  – input_tokens = n_tokens
  – spawn_and_wait_for_all(stage_task)
  • The first task to process the first token in the first stage
Inside pipeline.cpp: stage_task

- A class derived from task class
  - Kernel of the pipeline execution. Overrides the execute() method of task
  - Sketch code for the case in which the filter is parallel:

```cpp
task * stage_task::execute()
{
    if (first_entry && --input_tokens > 0){
        spawn_sibling(stage_task); first_entry=false}
    item=my_filter.function(/*previous*/item); //executes the stage
    my_filter=my_filter->next_filter_in_pipeline;
    if(my_filter){ //an intermediate filter in the pipeline
        recycle_as_continuation(); next=this; //Scheduler bypass
        return next;
    } else { //last filter in the pipeline has completed an item
        input_tokens++; // so start a new token (item)
        if(input_tokens==1) spawn_sibling(stage_task);
        return NULL
    }
}
```

Agenda

- Introduction.
  - Work Stealing. Installing and using

- Basic Classes
  - parallel_for, reduce, do, pipeline,…

- Advanced Classes
  - containers, locks, atomics, alloc, graphs,…

- Implementation details
  - Dealing with tasks. Examples
    - Recipes and advices
      - Research results
Pieces of advice

• Enough workload
  – To benefit from parallel_for
    • The loop has to take at least \(10^6\)ck
    • In a 2GHz processor → at least 500 microseconds

• Avoid blocking the thread
  – If a task blocks the thread (IO, mutex, …) the pending tasks are also blocked
  – If there is no oversubscription, the core gets idle

• TBB is interoperable (with Pthreads, OpenMP, etc)

Mixing with OpenMP

• Nest a TBB region inside a parallel OpenMP one
  – Create a task_scheduler_init inside the parallel region
  – Example:

```c
void OpenMP_Calls_TBB( int n ) {
    #pragma omp parallel
    {
        task_scheduler_init init;
        #pragma omp for
        for( int i=0; i<n; ++i ) {
            ...can use class task or
            Threading Building Blocks algorithms here
            ...
        }
    }
}
```
TBB vs. OpenMP

• Advantages of TBB over OpenMP:
  – TBB is a library, so it doesn’t need any special compiler support
  – TBB does not require programmer to worry about loop scheduling policies (static, dynamic, and guided)
  – Thanks to generic programming, paralle_reduce works on any type, unlike OpenMP that reduction is only applicable on built-in types
  – TBB provides thread-safe containers
  – TBB implements nested parallelism, not supported by all OpenMP implementations
  – OpenMP is mainly designed to parallelize loops and does not perform well on task-base parallelism

27/01/12  TBB tutorial. Rafael Asenjo.  Source: Navid Abbaszadeh

TBB vs. OpenMP

• Advantages of OpenMP over TBB
  – OpenMP is much simpler and has an easier learning curve
  – Minimal changes to serial program can be made incrementally until obtaining desired performance, unlike TBB that needs major changes
  – OpenMP is an open standard (TBB is public domain)
Agenda

✓ Introduction.
  - Work Stealing. Installing and using
✓ Basic Classes
  - parallel_for, reduce, do, pipeline,…
✓ Advanced Classes
  - containers, locks, atomics, alloc, graphs,…
   ➢ Implementation details
     ✓ Dealing with tasks. Examples
     ✓ Recipes and advices
      ➢ Research results
       ➢ Pipeline
        - Wavefront
        - Scheduler for heterogeneous CPU-GPU system

The PARSEC suite

- Princeton Application Repository for Shared-Memory Computers (http://parsec.cs.princeton.edu)
  - Multithreading emerging workloads
  - Diverse enough: RMS, streaming, multimedia, engineering…

<table>
<thead>
<tr>
<th>Program</th>
<th>Application Domain</th>
<th>Parallelization Model</th>
<th>Data Usage Sharing</th>
</tr>
</thead>
<tbody>
<tr>
<td>blackscholes</td>
<td>Financial Analysis</td>
<td>data-parallel</td>
<td>high</td>
</tr>
<tr>
<td>bodytrack</td>
<td>Computer Vision</td>
<td>data-parallel</td>
<td>high</td>
</tr>
<tr>
<td>cannonball</td>
<td>Engineering</td>
<td>unstructured</td>
<td>high</td>
</tr>
<tr>
<td>dsehup</td>
<td>Enterprise Storage</td>
<td>pipeline</td>
<td>high</td>
</tr>
<tr>
<td>facesim</td>
<td>Animation</td>
<td>data-parallel</td>
<td>medium</td>
</tr>
<tr>
<td>ferret</td>
<td>Similarity Search</td>
<td>pipeline</td>
<td>medium</td>
</tr>
<tr>
<td>fluidanimate</td>
<td>Animation</td>
<td>data-parallel</td>
<td>low</td>
</tr>
<tr>
<td>freqmine</td>
<td>Data Mining</td>
<td>data-parallel</td>
<td>high</td>
</tr>
<tr>
<td>streamcluster</td>
<td>Data Mining</td>
<td>data-parallel</td>
<td>medium</td>
</tr>
<tr>
<td>sveptions</td>
<td>Financial Analysis</td>
<td>data-parallel</td>
<td>low</td>
</tr>
<tr>
<td>vips</td>
<td>Media Processing</td>
<td>data-parallel</td>
<td>medium</td>
</tr>
<tr>
<td>zinc</td>
<td>Media Processing</td>
<td>pipeline</td>
<td>medium</td>
</tr>
</tbody>
</table>
Programming with TBB

- Two parallel pipeline applications:
  - Similarity search: ferret
  - Data compression: dedup

Similarity search: ferret

- Two serial stages and 4 middle parallel stages
  - Read a set of key images
  - Return the list of the most similar images in the Data Base
ferret performance

• Target machine: HP9000 Superdome SMP (picasso)
  – 64 dual-core Itanium2, 1.6GHz, 380GB MP, 9MB L3
• Input data set: “native”
  – 3,500 img. Queries (22M) / database 59,695 (78M) / top 50
  – gcc-serial=760 sec.
  – icc-serial=440 sec.
  – Parallel versions compiled with icc 10.1.018 and –O3

Efficiency problems:
  – Stage load imbalance and Input bottleneck
Data compression: dedup

- Main differences
  - S2 generate more output items than there were input
  - Some items may skip stage S4

![Diagram of data compression process]

27/01/12 TBB tutorial. Rafael Asenjo.

dedup performance

- In Superdome
- Input=“native”
  - 1 iso file = 672 MB
  - gcc-serial= 91.18 sec.
  - icc-serial= 89.37 sec
  - Parallel versions compiled with icc 10.1.018 and –O3

![Graph showing dedup performance]
dedup performance

Same problems: load imbalance and output bottleneck

Scalability issues and solutions

• Scalability gated by
  – Load imbalance for small number of threads
  – I/O bottleneck for larger number of threads

• Solutions
  – Load imbalance
    • Collapsing all the parallel stages into one (not general)
    • Oversubscription: semi-dynamic scheduling
    • Work stealing: dynamic scheduling
  – I/O bottleneck
    • Parallel I/O
Collapsing

• Horizontal vs. vertical configurations
• Only if all intermediate stages are parallel

The arrows translate into overhead

8 cores in both cases!

Semi-dynamic scheduling

• Oversubscription (semi-dynamic)
  – More threads than available cores
  – The # threads per stage is static
  – The OS dynamically assign the threads to available cores

  – Time slicing overheads:
  – Recommended for ferret and dedup
    • # of parallel threads per stage (c) = # cores
    • # threads is 4 (ferret) or 3 (dedup) times larger than # cores
Dynamic scheduling

• Work stealing
  – Only one thread per core (no time-slicing overheads)
  – Several work units (tasks) assigned to each thread
  – When a thread completes it tasks, steal work to other threads

• Implementations
  – Cilk: non trivial to combine C code and Cilk code
  – X10 (XWS): translation of the whole application to Java
  – Intel Threading Building Blocks (TBB)
    • C++ library (generic programming like in STL)
    • Several tasks per thread. Internal task scheduler

Experimental results

• Proposed solutions
  – Collapsing
  – Oversubscription
  – Work stealing

• Oversubscription

<table>
<thead>
<tr>
<th>Threads</th>
<th>6-stages Pthreads not-oversubscribed</th>
<th>6-stages Pthreads oversubscribed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># cores</td>
<td>Time (sec.)</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>370,68</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>186,69</td>
</tr>
<tr>
<td>18</td>
<td>18</td>
<td>94,87</td>
</tr>
<tr>
<td>34</td>
<td>34</td>
<td>47,95</td>
</tr>
<tr>
<td>66</td>
<td>66</td>
<td>38,41</td>
</tr>
</tbody>
</table>
Experimental results

- Using the ntokens suggested by our model
- TBB independent of the number of stages
- I/O bottleneck: 33 sec of input serial time
  - Amdahl’s law → maximum speed-up is 13.24

<table>
<thead>
<tr>
<th># cores</th>
<th>6 Stages TBB Speed-up</th>
<th>Efficiency</th>
<th>3 Stages TBB Speed-up</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>437,65</td>
<td>1,00</td>
<td>99,85%</td>
<td>442,88</td>
</tr>
<tr>
<td>1</td>
<td>221,13</td>
<td>1,98</td>
<td>98,81%</td>
<td>223,5</td>
</tr>
<tr>
<td>2</td>
<td>112,59</td>
<td>3,88</td>
<td>97,03%</td>
<td>113,58</td>
</tr>
<tr>
<td>8</td>
<td>58,5</td>
<td>7,47</td>
<td>93,38%</td>
<td>59,56</td>
</tr>
<tr>
<td>16</td>
<td>35,87</td>
<td>12,18</td>
<td>76,14%</td>
<td>36,98</td>
</tr>
<tr>
<td>32</td>
<td>33,14</td>
<td>13,19</td>
<td>41,21%</td>
<td>33,42</td>
</tr>
<tr>
<td>64</td>
<td>35,66</td>
<td>12,25</td>
<td>19,15%</td>
<td>36,44</td>
</tr>
</tbody>
</table>

TBB vs. Pthreads

- Improvement of 6-st. TBB over 6-st. Pthreads (ferret)
  - Improvement of 3-st. TBB over 3-st. Pthreads (ferret)
  - Improvement of 3-st. TBB over 3-st. Pthreads (dedup)
I/O optimizations

- 4-stages ferret
  - S1: serially read file name
  - S2: read file content in parallel
  - S3: crunching numbers
  - S4: serial output
- In Pthreads
  - 4 threads for S2
- In TBB
  - Dynamic # tasks in S2
- Not trivial for dedup

Summary of results

- Advantages of using TBB to implement work stealing
  - Productivity (easier to write, # stages and load balance not a prob.)
  - Behaves near optimally
  - Negligible overheads (measured with Vtune)
- Once load balancing is resolved I/O becomes the next bottleneck
- More in:
  - Analytical Modeling of Pipeline Parallelism, PACT’09
Agenda

✓ Introduction.
  – Work Stealing. Installing and using
✓ Basic Classes
  – parallel_for, reduce, do, pipeline,…
✓ Advanced Classes
  – containers, locks, atomics, alloc, graphs,…
➢ Implementation details
  ✓ Dealing with tasks. Examples
  ✓ Recipes and advices
  ➢ Research results
    ✓ Pipeline
    ➢ Wavefront
      – Scheduler for heterogeneous CPU-GPU system

Wavefront problems

• Wavefront is a programming pattern
  – Appears in important scientific applications
    • Dynamic programming or sequence alignment.
    • Checkerboard, Floyd, Financial, H.264 video compression.

• 2D wave problem

1 for (i=1; i<n; i++)
2   for (j=1; j<n; j++)
3   A[i,j] = foo(gs, A[i,j], A[i-1,j], A[i,j-1]);

• gs (grain size): tune the workload in our experiments.
2D Wavefront problem

\[ A[i,j] = \text{foo}(gs, A[i,j], A[i-1,j], A[i, j-1]); \]

Task based implementations: pseudocode

Each element computation is done by a task
OpenMP 3.0

```c
void Operation(int i, int j)
{
  int gs;
  bool ready;

  A[i][j] = foo(gs, A[i][j], A[i-1][j], A[i][j-1]);
  if (j<n-1) {
    omp_set_lock(&locks[i][j+1]);
    ready = --counter[i][j+1]==0;
    omp_unset_lock(&locks[i][j+1]);
    if (ready){
      #pragma omp task
      Operation(i, j+1);}
  if (i<n-1){
    omp_set_lock(&locks[i+1][j]);
    ready = --counter[i][j+1]==0;
    omp_unset_lock(&locks[i+1][j]);
    if (ready){
      #pragma omp task
      Operation(i+1, j);}
  }
}
```

OpenMP_v1 (critical)

OpenMP_v2 (locks)

Memory compsumption due to the matrix of locks

TBB_v1 implementation

```c
Class Operation: public TBB::task
{
  int i, j, gs;
  public:
    Operation(int i_, int j_): i(i_), j(j_) {}
  task * execute();
};
TBB::task * Operation::execute()
{
  A[i][j] = foo(gs, A[i][j], A[i-1][j], A[i][j-1]);
  if (i<n-1) //There is south neighbor
    if (--counter[i+1][j]==0)
      spawn( ..... Operation(i+1, j) );
  if (j<n-1) //There is east neighbor
    if (--counter[i][j+1]==0)
      spawn( ..... Operation(i, j+1) );
```
TBB_v2 implementation

```cpp
1 class MyBody{
2 public:
3 MyBody(){};
4 void operator() (block & b, tbb::parallel_do_feeder
5 <block> & feeder) const{
6 int i = b.first;
7 int j = b.second;
8 if (i<n && j<n) {
9     A[i][j] = foo(gs, A[i][j], A[i-1][j], A[i][j-1]);
10    if (i<n-1 && --counter[i+1][j]==0)
11        feeder.add(block(i+1, j));
12    if (j<n-1 && --counter[i][j+1]==0)
13        feeder.add(block(i, j+1));
14 }
15 }
```

Cilk Implementation

```cpp
void Operation(int i, int j)
1 {
2     int gs;
3     bool ready_e, ready_s;
5     if (j<m-1) {
6         pthread_mutex_lock(&locks[i][j+1]);
7         counter[i][j+1]--;
8         ready_e = (counter[i][j+1]==0);
9         pthread_mutex_unlock(&locks[i][j+1]);
10    }
11    if (ready_e)
12        cilk_spawn Operation(i, j+1);
13 }
14 }
15 }
16 if (i<m-1) {
17     pthread_mutex_lock(&locks[i+1][j]);
18     counter[i+1][j]--;
19     ready_s = (counter[i+1][j]==0);
20     pthread_mutex_unlock(&locks[i+1][j]);
21    }
22    if (ready_s)
23        cilk_spawn Operation(i, j+1);
24 }
25 }
26 if (ready_e || ready_s)
27    cilk_sync;
```
CnC particularities
- Atomic<int>
- Operation Collection
- Element Tag

```cpp
1 // Declarations. Tag collection to control execution
2 < par ElementTag >
3 // Step prescription: for each ElementTag instance
4 // we control an step exec.
5 <ElementTag>:: (Operation)
6 // Step execution: a step may produce a new ElementTag
7 (Compute) -> <ElementTag>
8 int Operation::execute(const par & t, wave & c ) const
9 {
10    int i = t.first;
11    int j = t.second;
12    int gs;
13
15    if (i < n-1)
16       if (--counter[i+1][j] == 0)
17          c.ElementTag.put(par(i+1,j));
18    if (j < n-1)
19       if (--counter[i][j+1] == 0)
20          c.ElementTag.put(par(i, j+1));
21    return CnC::CNC_Success;
22 }
```

Experimental results

- Two Quad-Core Intel Xeon CPU X5355 at 2.66 GHz.
- Running example case study. Matrix size = 1,000 x 1,000

Measures:
- Speedup (with respect to the fastest sequential code)
- Intel Vtune profiler
  - Callgraph: number callers, self time, wait time..
  - More time-consuming library functions

Two set of experiments

1. Constant task granularity (balanced workload).
   - Fine grain (200 FLOP). Medium (2,000 FLOP) and coarse (20,000 FLOP)
2. Variable grain size (unbalanced workload).
   - Fine grain (200..800 FLOP). Medium (800..2,000 FLOP) and coarse (2,000..20,000 FLOP)
1. Constant granularity: Speedups

**Notes:**
- Cilk does not complete some times.
- Similar behavior TBB_v1 and TBB_v2.
- OpenMP has the worst scalability.
- CnC is between TBB and OpenMP.

OpenMP: Most time consuming library functions (fine granularity):
- task (omp_task, omp_task_alloc): associated to spawn a new task.
- critical (omp_critical, omp_end_critical): associated to the critical directive.
- lock (omp_set_lock, omp_unset_lock): associated to the locks.
1. Constant granularity: Overheads

TBB: Most time consuming library functions (fine granularity):
- spawn: invoked each time a new task is spawned
- get_task: called after completing the execution of a previous task.
- allocate: selects the best memory allocation mechanism.

CnC: Most time consuming library functions (fine granularity):
- Helper functions called by the c.ElementTag.put() method.
2. Variable task granularity: Speedups.

![Graphs showing speedups for different task granularities: fine, medium, coarse.]

Notes:
- TBB_v1 slightly better than v2 for fine.
- OpenMP_v1: worst scalability
- OpenMP_v2: better now for fine and medium
- CnC is between OpenMP_v1 and v2
- Similar overhead results.

Optimizations: reducing overheads in TBB

```c
Task_Body(); // Task's work

if (j<n-1){ // There is east neighbor
    if ( -- counters[i][j+1] == 0 )
        recycle_into_east = true;

    if (i<n-1){ // There is south neighbor
        if ( -- counters[i+1][j] == 0 )
            if ( recycle_into_east )
                recycle_into_south = true;
            else
                spawn(i+1,j));

        if ( recycle_into_east )  //Recycle this into east
            recycle_as_child_of();
        else if(recycle_into_south) //Recycle this into south
            recycle_as_child_of();
        else
            return NULL; // There is no neighbor task ready

} else if(recycle_into_south)
    return this;
```

Exploiting task passing mechanism available in TBB

TBB_v4

Number of spawns: O(n^2) → O(n)
Optimizations: reducing overheads in TBB

TBB_v3 is similar to TBB_v4 but without exploiting cache locality

- Scalability improvement: near 3x
- Reduction overhead: near 7x
- Cache improvement: near 4x

```cpp
#include "wavefront.h"

void Operation::execute()
{
    int i = getFirst();
    int j = getSecond();
    A[i][j] = foo(A[i][j], A[i-1][j], A[i][j-1], A[i-1][j-1]);
}

int main()
{
    // waveform_init(); // Initialise TBB and vars
    // waveform_run(); // execute the waveform code

    int main()
    {
        atomic<int>* counters;

        for (i=0; i<4; i++)
            for (j=0; j<4; j++)
                if (i == j)
                    counters[i][j] = 1;
                else
                    counters[i][j] = 0;

        TBB_task_scheduler_init();
        TBB_spawn_root_and_wait ( Operation(i, j) );
    }

    TBB tutorial. Rafael Asenjo.
}
```
Wavefront Template

**Definition file**

// Data grid
[0:n-1,0:n-1]

// Task grid
[1:n-1, 1:n-1]

// Indices
<i, j>

// Dependency vectors
[1:n-2, 1:n-2] -> (0,1); (1,0)
[n-1, 1:n-2] -> (0,1)
[1:n-2, n-1] -> (1,0)

// (Optional): counter values
[1,1] = 0
[1, 2:n-1] = 1
[2:n-1, 1] = 1
[2:n-1, 2:n-1] = 2

**Main file**

```
#include "wavefront.h"

void Operation::ExecuteTask()
{
    int i = GetFirst();
    int j = GetSecond();
    A[i][j] = foo(gs, A[i][j], A[i-1][j], A[i][j-1]);
}

int main()
{
    ....

    wavefront_init(); // Initialize TBB and vars
    wavefront->run(); // Execute the wavefront code
    ....
}
```
Wavefront Template

Speedups of TBB manual vs. Template in real problems codes

1. Programmer specifies: i) definition file with dependence pattern information and ii) the task function.
2. The template is a productive tool (50% less programming effort) with low overhead costs (less than 5%).
Sumary of wavefront results

- TBB features allow more efficient implementations:
  - Atomic capture and Task recycling
- OpenMP greatly benefit from atomic capture (supported in OpenMP 3.1 gcc4.7 July 2011)
- CnC is a valid alternative, but for coarse grain
- Cilk++ implementation is not robust enough
- Future work
  - Experiments with a major number of cores
  - Hybrid wavefront program in GPU and CPU.
- More in:
  - High-level template for the task-based parallel wavefront pattern, HiPC 2011
  - A case study of the task-based parallel wavefront pattern, ParCo 2011

Agenda

- Introduction.
  - Work Stealing. Installing and using
- Basic Classes
  - parallel_for, reduce, do, pipeline,…
- Advanced Classes
  - containers, locks, atomics, alloc, graphs,…
- Implementation details
  - Dealing with tasks. Examples
  - Recipes and advices
- Research results
  - Pipeline
  - Wavefront
  - Scheduler for heterogeneous CPU-GPU system
TBB for heterogeneous

• In presence of GPUs
  – Idea:
    • A dedicated thread wrapping the GPU
    • This thread has the work-stealing scheduler to feed the GPU
    • The GPU notifies this thread when idle

TBB for heterogeneous

• In presence of GPUs
  – Higher level approach →
    • TBB pipeline template:
      – A pipeline stage served by a GPU
        » Should be the bottleneck stage
    • TBB Parallel_for template:
      – Adjust the grain size of the blocked_range iteration space for the tasks that will be mapped in the GPU
parallel_for with GPU

• Relying in a two-stages pipeline:
  – 1\textsuperscript{st} serial stage: generates chunks for the next stage
  – 2\textsuperscript{nd} parallel stage: computes a chunk in CPU or GPU

• First approach: fixed chunk size

• Second approach: variable chunk size
parallel_for with GPU

- Third approach based on TBB flow_graphs
  - A token generator creates N (#cores + #GPUs) tokens
  - Token filler add information to the token
    - Range to compute and where to compute
    - A range sent to GPU also has a sub-range to be computed in CPU
  - Worker: send the range to GPU and/or CPU
    - The CPU is invigilating and helping the GPU
    - If the GPU comes to an end before the CPU, the token is released back with the remaining iterations that need re-schedule

Conclusions

- Portable and efficient tool to parallelize codes
- Specially useful for fine grain problems
- Interoperable with other parallelizing languages and tools
- Pipeline, containers and direct task support
- General programming and nested parallelism
- Cache conscious
- Drawbacks
  - More productive than Pthreads, but less than OpenMP
  - Requires C++ programming
Backup slides

• C++ concepts
  – Slides from: Arch D. Robison. Sr. Principal Engineer Performance Analysis and Threading (PAT). Intel Corporation, Champaign IL
    • C++ general programming concept and lambda notation

C++ Review

“Give me six hours to chop down a tree and I will spend the first four sharpening the axe.”

- Abraham Lincoln
C++ Review: Half Open Intervals

• STL usually specifies sequence as half-open interval \([first, last)\)
  – \(first \text{=} last \iff \text{empty interval}\)
  – \(last - first\) is size of interval

• If object \(x\) contains a sequence
  – \(x\).begin() denotes first element (or pointer to first element).
  – \(x\).end() denotes “one past last” element.

Example

```c++
void PrintContainerOfTypeX(const X& x) {
    for (X::iterator i = x.begin(); i != x.end(); ++i)
        cout << *i << endl;
}
```

C++ Review: Function Template

• Type-parameterized function.
  – Strongly typed.
  – Obeys scope rules.
  – Actual arguments evaluated exactly once.
  – Not redundantly instantiated.

```c++
template<typename T>
void swap(T& x, T& y) {
    T z = x;
    x = y;
    y = z;
}

void reverse(float* first, float* last) {
    while (first < last - 1)
        swap(*first++, *--last);
}
```

[See page 166 for the continuation of the text.]
Genericity of swap

```cpp
template<typename T>
void swap( T& x, T& y ){
    T z = x;                              // Construct z
    x = y;                               // Assignment
    y = z;                               // Assignment
}
```

Requirements for T

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>T(const T&amp;)</code></td>
<td>Copy constructor</td>
</tr>
<tr>
<td><code>void T::operator=(const T&amp;)</code></td>
<td>Assignment</td>
</tr>
<tr>
<td><code>~T()</code></td>
<td>Destructor</td>
</tr>
</tbody>
</table>

C++ Review: Template Class

- Type-parameterized class

```cpp
template<typename T, typename U>
class pair {
    public:
        T first;
        U second;
        pair( const T& x, const U& y ) : first(x), second(y) {}
    }
```

```cpp
pair<string, int> x;
x.first = "abc";
x.second = 42;
```

Compiler instantiates template `pair` with `T=string` and `U=int.`
C++ Function Object

- Also called a “functor”
- Is object with member operator().

```cpp
class LinearOp {
    float a, b;
public:
    float operator()( float x ) const { return a*x+b; }
    LinearOp( float a_, float b_ ) : a(a_), b(b_) {} }
```

```cpp
LinearOp f(2,5);
float y = f(3); // Could write this line as y = f.operator()(3);
```

Template Function + Functor = Flow Control

```cpp
template<typename I, typename Functor>
void ForEach( I lower, I upper, const Functor& f ) {
    for( I i=lower; i<upper; ++i )
        f(i);
}
```

```cpp
class Accumulate {
    float& acc;
    float* src;
public:
    Accumulate( float& acc_, float* src_ ) : acc(acc_), src(src_) {}
    void operator()( int i ) const { acc += src[i]; }
};
```

```cpp
float Example() {
    float a[4] = {1,3,9,27};
    float sum = 0;
    ForEach( 0, 4, Accumulate(sum,a) );
    return sum;
}
```

Pass functor to template function.
So Far

- Abstract control structure as a template function.
- Encapsulate block of code as a functor.
- Template function and functor can be arbitrarily complex.

Recap: Capturing Local Variables

- Local variables were captured via fields in the functor.

```cpp
class Accumulate {
    float& acc;
    float* src;
public:
    Accumulate(float& acc_, float* src_) : acc(acc_), src(src_) {}
    void operator()(int i) const {
        acc += src[i];
    }
};

float Example() {
    float a[4] = {1,3,9,27};
    float sum = 0;
    ForEach(0, 4, Accumulate(sum,a));
    return sum;
}
```
Array Can Be Captured as Pointer Value

```cpp
class Accumulate {
    float& acc;  // Field for capturing a declared as a pointer.
    float* src;

public:
    Accumulate( float& acc_, float* src_ ) :
        acc(acc_), src(src_) {}
    void operator()( int i ) const {
        acc += src[i];
    }
};

float Example() {
    float a[4] = {1,3,9,27};
    float sum = 0;
    ForEach( 0, 4, Accumulate(sum,a) );
    return sum;
}
```

An Easier Naming Scheme

- Name each field and parameter after the local variable that it captures.

```cpp
class Accumulate {
    float& sum;  // a implicitly converts to pointer
    float* a;

public:
    Accumulate( float& sum_, float* a_ ) :
        sum(sum_), a(a_) {}
    void operator()( int i ) const {
        sum += a[i];
    }
};

float Example() {
    float a[4] = {1,3,9,27};
    float sum = 0;
    ForEach( 0, 4, Accumulate(sum,a) );
    return sum;
}
```

This is still tedious mechanical work. Can we make the compiler do it for us?
C++0x Lambda Expressions

• Part of C++0x draft
• Implemented in Intel® C++ Compiler 11.0.
  – Linux* OS: -std=c++0x
  – Windows* OS: /Qstd:c++0x
• Creates functor that captures local variables.

```
float Example() {
    float a[4] = {1,3,9,27};
    float sum = 0;
    ForEach( 0, 4, [&]( int i ) {sum += a[i]; } );
    return sum;
}
```

Compiler automatically defines custom functor type tailored to capture `sum` and `a`.

[[&]] introduces lambda expression that constructs instance of functor.

Parameter list and body for functor::operator()

Lambda Syntax

```
[capture_mode] (formal_parameters) -> return_type {body}
```

Can omit if there are no parameters and return type is implicit. Can omit if return type is void or code is “return expr;”

[&] ⇒ by-reference
[=] ⇒ by-value
[] ⇒ no capture

Examples

```
[&](float x) {sum+=x;}

[&](return *p++;

[=](float x) {return a*x+b;}
```

Not covered here: how to specify capture mode on a per-variable basis.
Notes

• Compiler creates a unique anonymous functor type for each lambda expression.
• Therefore lambda expressions are generally useful only as arguments to template functions or with C++0x auto keyword.

```cpp
template< typename F >
void Eval( const F & f ) {
    f();
}

void Example1() {
    Eval( []{ printf("Hello, world\n"); } );
}

void Example2() {
    auto f = []{ printf("Hello, world\n"); };
    f();
}
```

Template deduces functor’s type instead of specifying it.

Expression []{...} has anonymous type.

Compiler deduces type of f from right side expression.
Implemented in Intel® C++ Compiler 11.0.