Concurrent Programming

Session 10: Intel Threading Building Blocks (TBB)

Computer Engineering Department
Iran University of Science and Technology
Tehran, Iran

Lecturer: Navid Abbaszadeh
Distributed Systems Lab.
Computer Engineering Department,
Iran University of Science and Technology,
nabbaszadeh@comp.iust.ac.ir

What is TBB?

- Intel® Threading Building Blocks is a C++ runtime library that abstracts the low-level threading details necessary addressing multicore performance
- It uses C++ templates and coding style for implementation
- It requires fewer lines of code to achieve parallelism compared to using threads explicitly
- The applications are portable across platforms.
- The library is scalable. That is, as more processor cores become available code need not be rewritten.

OS's Supported

- Microsoft Windows XP Professional
- Microsoft Windows Server 2003
- Microsoft Windows Vista
- Red Hat Enterprise Linux 3, 4 and 5
- Red Hat Fedora Core 4, 5 and 6
- Asianux 2.0
- Red Flag DC Server 5.0
- Haansoft Linux Server 2006
- Miracle Linux v4.0
- SuSE Linux Enterprise Server (SLES) 9 and 10
- SGI Propack 4.0 & SGI Propack 5.0
- Mandriva/Mandrake Linux 10.1.06
- Turbolinux GreatTurbo Enterprise Server 10 SP1

Compilers Supported

- Microsoft Visual C++ 7.1 (Microsoft Visual Studio .NET 2003, Windows systems only)
- Microsoft Visual C++ 8.0 (Microsoft Visual Studio 2005, Windows systems only)
- Intel® C++ Compiler 9.0 or higher (Windows and Linux systems)
- Intel® C++ Compiler 9.1 or higher (Mac OS X systems)
- For each supported Linux operating system, the standard gcc version provided with that operating system is supported, including: 3.2, 3.3, 3.4, 4.0, 4.1
- For each supported Mac OS X operating system, the standard gcc version provided with that operating system is supported, including: 4.0.1 (Xcode tool suite 2.2.1 or higher)

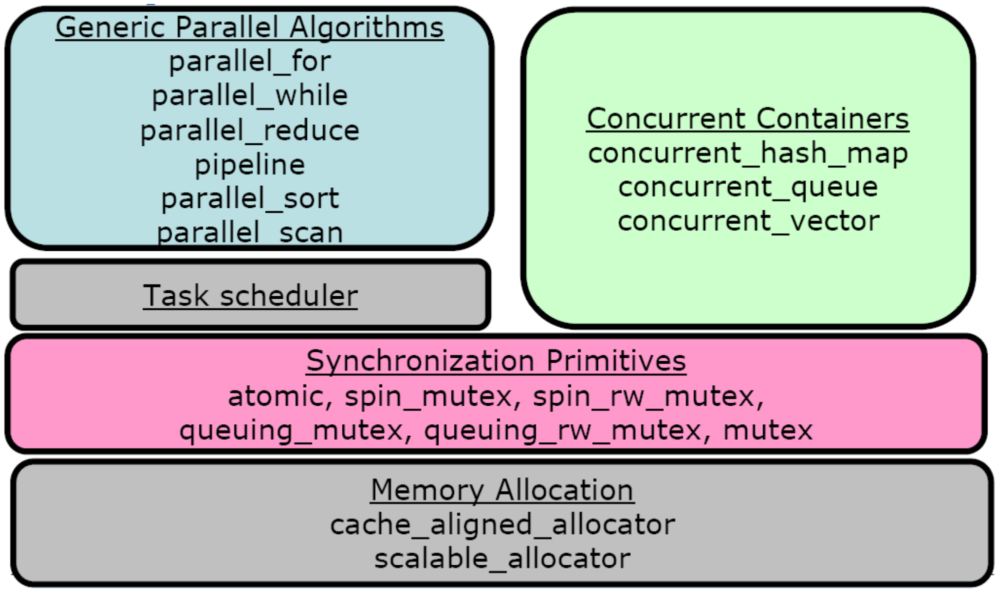
Why do we need this?

- Gaining performance (in a single application) from multiple cores requires concurrent/parallel programming.
- Concurrent programming introduces the issues of race conditions and deadlocks.
- Concurrent programming for scalability is a difficult task.
- Not all threading interfaces work on all platforms (POSIX threads).
- Programming with threads introduces another dimension of complexity.

Limitations

- TBB is not recommended for:
  - I/O bound processing
  - Hard real time processing
- TBB is not a silver bullet for all multi-threaded applications. It's a tool that heads us in the correct direction, but is not optimum.

Components



Runtime library initialization and termination

- The scheduler is initialized by task_scheduler_init object's constructor and it is destroyed by its destructor

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

int main() {
    task_scheduler_init init;
     ...
    return 0;
}
```

Runtime library initialization and termination(cont')

- The constructor of task_scheduler_init can be given a parameter :
  - Task_scheduler::automatic, which is the same as not specifying
  - Task_scheduler::deffered, which defers the initialization until
  - method task_scheduler_init::initialize(n) is called.
  - A positive integer specifying the number of threads to use

TBB Task Scheduler

- Automatically balance the load across processors
- Schedule tasks to exploit the natural cache locality of applications
- Avoid the over-subscription of resources that often comes when composing applications from threaded components.

Generic Parallel algorithms

- Parallel_for
- Parallel_reduce
- Parallel_sort
- pipeline

Parallel_for

- Serial code
- Assumption : iterations of loop are independant

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

Parallel_for (cont')

```
#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)
    {}
};
```

Parallel_for (cont')

- Parallelized version of code
- Automatic grain-size

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

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

Parallel_for (cont')

- Parallelized version of code
- Explicit grain-size

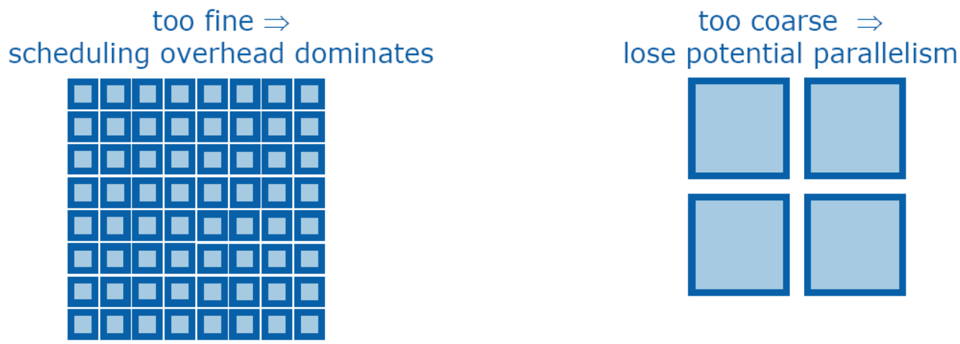
```
#include "tbb/parallel_for.h"

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

Grain Size

- Part of parallel_for, not the task scheduler.
- Specifies the number of iterations for a reasonable size chunk of data to deal out to the processor.
- Optimum value depends on the problem. That is, do some benchmarking.

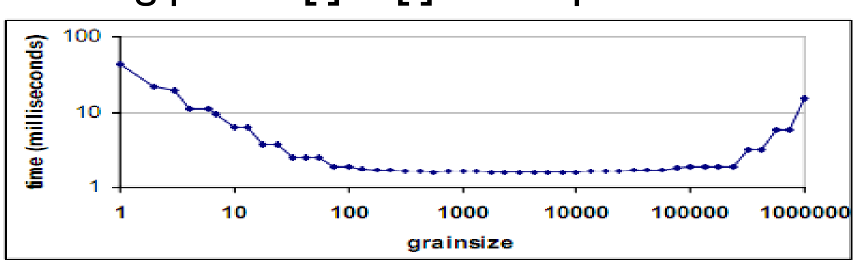
Tuning Grain Size

- When in doubt, err on the side of making it a little too large, so that performance is not hurt when only one core is available.



Tuning Grain Size

- A rule of thumb is that grainsize iterations of operator() should take at least 10,000-100,000 instructions to execute
- You do not have to set the grainsize too precisely.
- execution time versus grainsize, based on the floating point a[i]=b[i]*c computation



Parallel_reduce

- Serial code

```
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_reduce (cont')

- Parallelized code

```
float ParallelSumFoo( const float a[], size_t n
) {
    SumFoo sf(a);

parallel_reduce(blocked_range<size_t>(0,n),
sf, auto_partitioner() );
    return sf.my_sum;
}
```

Parallel_reduce (cont')

```
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; }
```

Parallel_reduce (cont')

```
 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)
    {}
}; // end of class sumFoo
```

Parallel_sort

- Performs an unstable sort of sequence [begin1, end1).
- The sort is deterministic
- parallel_sort is comparison sort with an average time complexity of O(N log (N))
- Requirements on Value Type T of RandomAccessIterator for parallel_sort :
  - void swap(T& x, T& y)
  - bool Compare::operator()(const T& x, const T& y)

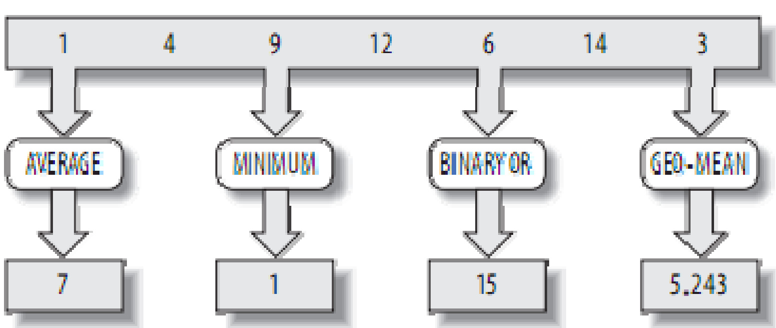
Parallel_sort (cont')

```
#include "tbb/parallel_sort.h"
#include <math.h>
using namespace tbb;

const int N = 100000;
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>());
}
```

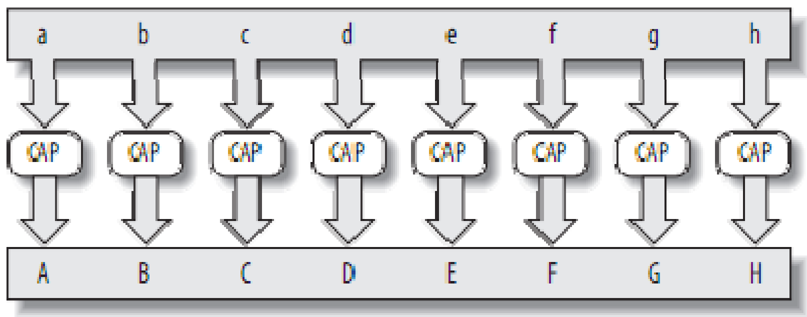
Getting Parallel

- There are two main methods for decomposing a sequential program into a parallel program:
  - Functional decomposition - independent tasks that are doing different types of work are identified. These functionally distinct tasks are then executed concurrently.
  - Data decomposition - a single task performed on a large amount of data is split into independent tasks, each task processing a subset of the data.

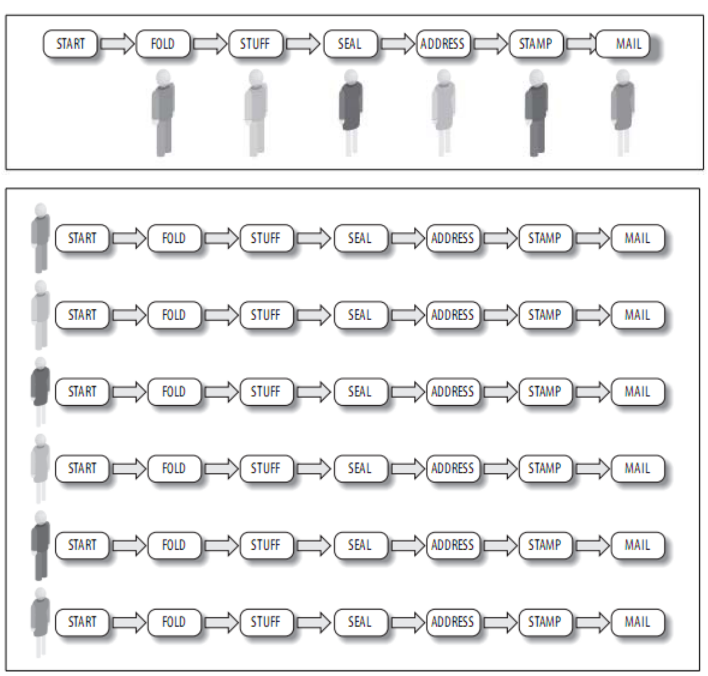
Functional(task) Decomposition



Data Decomposition



Another instance



Pipeline

- Pipelining is a common parallel pattern that mimics a traditional manufacturing assembly line
- An example : video processing
- The Intel® Threading Building Blocks classes pipeline and filter implement the pipeline pattern

Pipeline (cont')

```
// Create the pipeline
   tbb::pipeline pipeline;

   // Create file-reading writing stage and add it to the pipeline
   MyInputFilter input_filter( input_file );
   pipeline.add_filter( input_filter );

   // Create capitalization stage and add it to the pipeline
   MyTransformFilter transform_filter;
   pipeline.add_filter( transform_filter );

   // Create file-writing stage and add it to the pipeline
   MyOutputFilter output_filter( output_file );
   pipeline.add_filter( output_filter );

   // Run the pipeline
   pipeline.run( MyInputFilter::n_buffer );
```

Pipeline (cont')

```
// Filter that writes each buffer to a file.
class MyOutputFilter: public tbb::filter {
    FILE* my_output_file;
public:
    MyOutputFilter( FILE* output_file );
    /*override*/void* operator()( void* item );
};

MyOutputFilter::MyOutputFilter( FILE* output_file ) :
    tbb::filter(serial_in_order),
    my_output_file(output_file)
{ }

void* MyOutputFilter::operator()( void* item ) {
    MyBuffer& b = *static_cast<MyBuffer*>(item);
    fwrite( b.begin(), 1, b.size(), my_output_file );
    return NULL;
}
```

Parallel containers

- Containers provided by Intel® Threading Building Blocks offer a much higher level of concurrency, via one or both of the following methods:
  - Fine-grained locking. With fine-grain locking, multiple threads operate on the container by locking only those portions they really need to lock. As long as different threads access different portions, they can proceed concurrently.
  - Lock-free algorithms. With lock-free algorithms, different threads account and correct for the effects of other interfering threads.

Parallel containers (cont')

- Containers:
  - concurrent_hash_map
  - concurrent_vector
  - concurrent_queue
- Each container has its own constraints and defined functionality. For instance concurrent_queue does not guarantee a FIFO behaviour.

Mutual Exclusion (example)

```
Node* FreeList;
typedef spin_mutex FreeListMutexType;
FreeListMutexType FreeListMutex;

Node* AllocateNode() {
    Node* n;
    {
        FreeListMutexType::scoped_lock lock(FreeListMutex);
        n = FreeList;
        if( n )
            FreeList = n->next;
    }
    if( !n )
        n = new Node();
    return n;
}
```

Mutual Exclusion (terminology)

- Scalabililty
  - if the waiting threads consume excessive processor cycles and memory bandwidth, they reduce the speed of thread trying to execute code in critical section.
  - Non-scalable mutexes are faster under light contention
- Fairness
  - Fair mutexes avoid starving threads
  - Unfair mutexes are faster
- Recursive
  - A recursive mutex allows a thread that is already holding a lock on the mutex to acquire another lock on the mutex

Mutual Exclusion (kinds of mutexes)

Mutex	Scalable	Fair	Recursive
Mutex	OS-dependant	OS-dependant	No
Recursive_mutex	OS-dependant	OS-dependant	Yes
spin_mutex	No	No	No
queuing_mutex	Yes	Yes	No

Atomic Operations

- Fundamental Operations on a Variable x of Type atomic<T> are shown in the following table :

Code Snippet	meaning
= X	read the value of x
X =	write the value of x, and return it
x.fetch_and_store(y)	do y=x and return the old value of x
x.fetch_and_add(y)	do x+=y and return the old value of x
x.compare_and_swap(y,z)	if x equals z, then do x=y. In either case, return old value of x.

Timing

- Unlike some timing interfaces, tick_count is guaranteed to be safe to use across threads

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

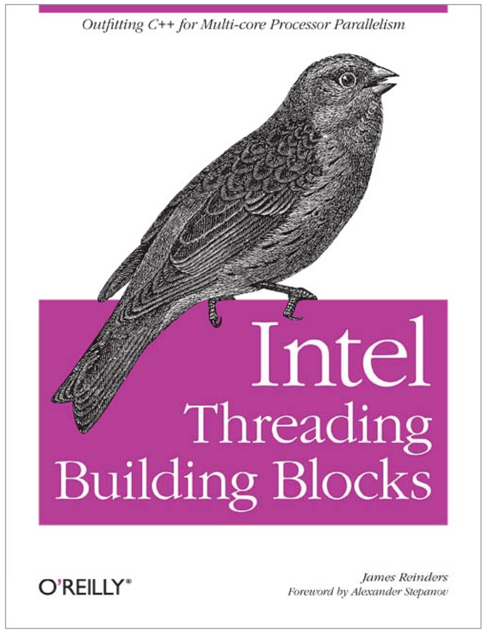
TBB VS openMP

- 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, Parallel_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, the feature that is not supported by all openMP implementations.
  - openMP is mainly designed to parallelize loops and does not perform well on task-base parallelism.

TBB VS openMP(cont')

- Advantages of openMP over TBB
  - openMP is much simpler and has a 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 has higher overhead in loops -for example, requiring grain size of ~100,000 for a loop

The Book



Thanks for you attention

Any questions?