Introduction to the Intel Threading Building Blocks

1. the Intel Threading Building Blocks (TBB)
   - programming computers with multicore processors
   - task based programming and work stealing

2. using TBB
   - our first program with TBB
   - compiling and running

3. using the `parallel_for`
   - raising complex numbers to a large power
   - speeding up the computations with `parallel_for`

4. using the `parallel_reduce`
   - an application of work stealing

MCS 572 Lecture 11
Introduction to Supercomputing
Jan Verschelde, 16 September 2016
Introduction to the Intel Threading Building Blocks

1. the Intel Threading Building Blocks (TBB)
   - programming computers with multicore processors
   - task based programming and work stealing

2. using TBB
   - our first program with TBB
   - compiling and running

3. using the `parallel_for`
   - raising complex numbers to a large power
   - speeding up the computations with `parallel_for`

4. using the `parallel_reduce`
   - an application of work stealing
We introduce programming tools for shared memory parallelism. Today we introduce a third tool:

- OpenMP: programming shared memory parallel computers,
- Pthreads: POSIX standard for Unix system programming,
- Intel Threading Building Blocks (TBB) for multicore processors.

Each tool has its own intrinsic merit, but the Intel TBB fits with the ubiquity of multicore processors: today every computer is parallel.
The Intel TBB is a library that helps you leverage multicore performance **without having to be a threading expert**.

The advantage of Intel TBB is that it works at a higher level than raw threads, yet does not require exotic languages or compilers.

The library differs from others in the following ways:
- TBB enables you to specify logical parallelism instead of threads;
- TBB targets threading for performance;
- TBB is compatible with other threading packages;
- TBB emphasizes scalable, data parallel programming;
- TBB relies on generic programming, (e.g.: use of STL in C++).

Open Source, download at
http://threadingbuildingblocks.org/.
Introduction to the Intel Threading Building Blocks

1. the Intel Threading Building Blocks (TBB)
   - programming computers with multicore processors
   - task based programming and work stealing

2. using TBB
   - our first program with TBB
   - compiling and running

3. using the `parallel_for`
   - raising complex numbers to a large power
   - speeding up the computations with `parallel_for`

4. using the `parallel_reduce`
   - an application of work stealing
Task based programming and work stealing

Tasks are much lighter than threads. On Linux,

- starting and terminating a task is about 18 times faster than starting and terminating a thread; and
- a thread has its own process id and own resources, whereas a task is typically a small routine.

The TBB task scheduler uses **work stealing** for load balancing.

In scheduling threads on processors, we distinguish between work sharing and work stealing:

- In work sharing, the scheduler attempts to migrate threads to under-utilized processors in order to distribute the work.
- In work stealing, under-utilized processors attempt to steal threads from other processors.
running an example program

To run the fractal example on kepler remotely:

1. Login as `ssh -X kepler.math.uic.edu -l userid`.

2. In the `.bashrc` file, add the line `export TBB_EXAMPLES_X_NOSHMEM=1` to avoid shared memory with X.

3. Go to the `/usr/local/tbb42_20131118oss` directory, and type `cd examples/task_priority/fractal`.

4. At the prompt type `./Fractal &` where the & makes that the job runs in the background.

5. To terminate the process, type `kill -9 %1`.
Introduction to the Intel Threading Building Blocks

1. the Intel Threading Building Blocks (TBB)
   - programming computers with multicore processors
   - task based programming and work stealing

2. using TBB
   - our first program with TBB
   - compiling and running

3. using the parallel_for
   - raising complex numbers to a large power
   - speeding up the computations with parallel_for

4. using the parallel_reduce
   - an application of work stealing
A class in C++ is like a struct in C for holding data attributes and functions (called methods).
the main function

```c
int main( )
{
    task_group tg;
    tg.run(say_hello("1")); // spawn 1st task and return
    tg.run(say_hello("2")); // spawn 2nd task and return
    tg.wait( );              // wait for tasks to complete
}
```

The `run` method spawns the task immediately, but does not block the calling task, so control returns immediately.

To wait for the child tasks to finish, the classing task calls `wait`.

Observe the syntactic simplicity of `task_group`. 
Introduction to the Intel Threading Building Blocks

1. the Intel Threading Building Blocks (TBB)
   - programming computers with multicore processors
   - task based programming and work stealing

2. using TBB
   - our first program with TBB
   - compiling and running

3. using the `parallel_for`
   - raising complex numbers to a large power
   - speeding up the computations with `parallel_for`

4. using the `parallel_reduce`
   - an application of work stealing
The makefile contains the following:

TBB_ROOT=/usr/local/tbb42_20131118oss

hello_task_group:
    g++ -I$(TBB_ROOT)/include \ 
    -L$(TBB_ROOT)/lib \ 
    hello_task_group.cpp \ 
    -o hello_task_group -ltbb
$ make hello_task_group

g++ -I/usr/local/tbb42_20131118oss/include \ 
    -L/usr/local/tbb42_20131118oss/lib \ 
    hello_task_group.cpp \ 
    -o hello_task_group -ltbb

$ ./hello_task_group
hello from task 2
hello from task 1
$
Introduction to the Intel Threading Building Blocks

1. the Intel Threading Building Blocks (TBB)
   - programming computers with multicore processors
   - task based programming and work stealing

2. using TBB
   - our first program with TBB
   - compiling and running

3. using the `parallel_for`
   - raising complex numbers to a large power
   - speeding up the computations with `parallel_for`

4. using the `parallel_reduce`
   - an application of work stealing
raising complex numbers to a large power

Consider the following problem:

Input: \( n \in \mathbb{Z}_{>0}, d \in \mathbb{Z}_{>0}, x \in \mathbb{C}^n \).
Output: \( y \in \mathbb{C}^n, y_k = x_k^d \), for \( k = 1, 2, \ldots, n \).

To avoid overflow, we take complex numbers on the unit circle.

In C++, complex numbers are defined as a template class. To instantiate the class \texttt{complex} with the type \texttt{double} we declare

```
#include <complex>

using namespace std;

typedef complex<double> dcmplx;
```
random complex doubles

```c
#include <cstdlib>
#include <cmath>

dcmplx random_dcmplx ( void );
// generates a random complex number
// on the complex unit circle

We compute \( e^{2\pi i \theta} = \cos(2\pi \theta) + i \sin(2\pi \theta) \), for random \( \theta \in [0, 1] \):

dcmplx random_dcmplx ( void )
{
    int r = rand();
    double d = ((double) r)/RAND_MAX;
    double e = 2*M_PI*d;
    dcmplx c(cos(e),sin(e));
    return c;
}
```
writing arrays

```c++
#include <iostream>
#include <iomanip>

void write_numbers ( int n, dcmplx *x );
// writes the array of n doubles in x

Observe the local declaration int i in the for loop, the scientific formatting, and the methods real() and imag():

void write_numbers ( int n, dcmplx *x )
{
    for(int i=0; i<n; i++)
        cout << scientific << setprecision(4)
             << "x[" << i << "] = ( " << x[i].real()
             << ", " << x[i].imag() << "]\n";
}
void compute_powers ( int n, dcmplx *x, 
               dcmplx *y, int d );
// for arrays x and y of length n, 
// on return y[i] equals x[i]**d

The plain for(int j loop avoids repeated squaring:

void compute_powers ( int n, dcmplx *x, 
               dcmplx *y, int d )
{
    for(int i=0; i < n; i++) // y[i] = pow(x[i],d);
    {
        // pow is too efficient
        dcmplx r(1.0,0.0);
        for(int j=0; j < d; j++) r = r*x[i];
        y[i] = r;
    }
}
command line arguments

$ /tmp/powers_serial
how many numbers ? 2
x[0] = ( -7.4316e-02 , 9.9723e-01)
x[1] = ( -9.0230e-01 , 4.3111e-01)
give the power : 3
x[0] = ( 2.2131e-01 , -9.7520e-01)
x[1] = ( -2.3152e-01 , 9.7283e-01)

$ /tmp/powers_serial 2 3 1
x[0] = ( -7.4316e-02 , 9.9723e-01)
x[1] = ( -9.0230e-01 , 4.3111e-01)
x[0] = ( 2.2131e-01 , -9.7520e-01)
x[1] = ( -2.3152e-01 , 9.7283e-01)

$ time /tmp/powers_serial 1000 1000000 0
real 0m20.139s
user 0m20.101s
sys 0m0.000s
the main program

```c
int main ( int argc, char *argv[] )
{
    int v = 1;       // verbose if > 0
    if(argc > 3) v = atoi(argv[3]);
    int dim;        // get the dimension
    if(argc > 1)
        dim = atoi(argv[1]);
    else
    {
        cout << "how many numbers ? ";
        cin >> dim;
    }
    // fix the seed for comparisons
    srand(20120203); // srand(time(0));
    dcmplx r[dim];
    for(int i=0; i<dim; i++)
        r[i] = random_dcmplx();
    if(v > 0) write_numbers(dim,r);
```

---

Introduction to Supercomputing (MCS 572)
int deg; // get the degree
if(argc > 1)
    deg = atoi(argv[2]);
else
{
    cout << "give the power : ";
    cin >> deg;
}
dcmplx s[dim];
compute_powers(dim, r, s, deg);
if(v > 0) write_numbers(dim, s);

return 0;
Introduction to the Intel Threading Building Blocks

1. the Intel Threading Building Blocks (TBB)
   - programming computers with multicore processors
   - task based programming and work stealing

2. using TBB
   - our first program with TBB
   - compiling and running

3. using the `parallel_for`
   - raising complex numbers to a large power
   - speeding up the computations with `parallel_for`

4. using the `parallel_reduce`
   - an application of work stealing
the speedup on kepler

$ time /tmp/powers_serial 1000 1000000 0

real 0m20.139s
user 0m20.101s
sys 0m0.000s

$ time /tmp/powers_tbb 1000 1000000 0

real 0m1.191s
user 0m35.170s
sys 0m0.043s

The speedup: \( \frac{20.139}{1.191} = 16.909 \) on two 8-core CPUs.
the class ComputePowers

class ComputePowers
{
    dcmplx *const c; // numbers on input
    int d;         // degree
    dcmplx *result; // output

public:
    ComputePowers(dcmplx x[], int deg, dcmplx y[]) :
        c(x), d(deg), result(y) { }

    void operator()
        ( const blocked_range<size_t>& r ) const
    {
        for(size_t i=r.begin(); i!=r.end(); ++i)
        {
            dcmplx z(1.0,0.0);
            for(int j=0; j < d; j++) z = z*c[i];
            result[i] = z;
        }
    }
};
#include "tbb/blocked_range.h"

template<typename Value> class blocked_range

A blocked_range represents a half open range \([i, j)\) that can be recursively split.

    void operator() ( const blocked_range<size_t>& r ) const 
    {
        for(size_t i=r.begin(); i!=r.end(); ++i) 
        { 

calling the `parallel_for`

```cpp
#include "tbb/tbb.h"
#include "tbb/blocked_range.h"
#include "tbb/parallel_for.h"
#include "tbb/task_scheduler_init.h"

using namespace tbb;

Two lines change in the main program:

```cpp
task_scheduler_init init(task_scheduler_init::automatic);
parallel_for(blocked_range<size_t>(0,dim),
            ComputePowers(r,deg,s));
```
Introduction to the Intel Threading Building Blocks

1. the Intel Threading Building Blocks (TBB)
   - programming computers with multicore processors
   - task based programming and work stealing

2. using TBB
   - our first program with TBB
   - compiling and running

3. using the `parallel_for`
   - raising complex numbers to a large power
   - speeding up the computations with `parallel_for`

4. using the `parallel_reduce`
   - an application of work stealing
an application of work stealing

from the Intel Threading Building Blocks Tutorial
what if no worker is available?

from the Intel Threading Building Blocks Tutorial
the class `SumIntegers`

class SumIntegers
{
    int *data;

public:
    int sum;
    SumIntegers ( int *d ) : data(d), sum(0) {}
    void operator() 
    ( const blocked_range<size_t>& r )
    {
        int s = sum; // must accumulate !
        int *d = data;
        size_t end = r.end();
        for(size_t i=r.begin(); i != end; ++i)
            s += d[i];
        sum = s;
    }
}
split and join methods

// the splitting constructor
SumIntegers ( SumIntegers& x, split ) :
    data(x.data), sum(0) {}

// the join method does the merge
void join ( const SumIntegers& x ) { sum += x.sum; }

int ParallelSum ( int *x, size_t n )
{
    SumIntegers S(x);

    parallel_reduce(blocked_range<size_t>(0,n), S);

    return S.sum;
}
int *d;
d = (int*)calloc(n,sizeof(int));
for(int i=0; i<n; i++) d[i] = i+1;

task_scheduler_init init
    (task_scheduler_init::automatic);
int s = ParallelSum(d,n);
Bibliography


The Intel TBB is our third tool for shared memory parallelism.

Exercises:

1. Modify the `hello world!` program with so that the user is first prompted for a name. Two tasks are spawned and they use the given name in their greeting.

2. Modify `powers_tbb.cpp` so that the $i$th entry is raised to the power $d - i$. In this way not all entries require the same work load. Run the modified program and compare the speedup to check the performance of the automatic task scheduler.