

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
 - environment variables, compiling, and running
- 3 using the `parallel_for`
 - raising complex numbers to a large power
 - speeding up the computations with `parallel_for`

MCS 572 Lecture 11
Introduction to Supercomputing
Jan Verschelde, 3 February 2012

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
- environment variables, compiling, and running

3 using the `parallel_for`

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

programming multicore processors

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.

Intel Threading Building Blocks

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
- environment variables, compiling, and running

3 using the `parallel_for`

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

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 dezon remotely:

- 1 Login as `ssh -X dezon.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 `examples/fractal` directory:
`/usr/local/tbb40_233oss/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
 - environment variables, compiling, and running
- 3 using the `parallel_for`
 - raising complex numbers to a large power
 - speeding up the computations with `parallel_for`

saying hello

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

class say_hello
{
    const char* id;
public:
    say_hello(const char* s) : id(s) { }
    void operator( ) ( ) const
    {
        printf("hello from task %s\n",id);
    }
};
```

A class in C++ is a like a struct in C
for holding data attributes and functions (called methods).

the main function

```
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
- **environment variables, compiling, and running**

3 using the `parallel_for`

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

environment variables

On dezon, add to the `.bashrc` file in your home directory:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/\
tbb40_233oss/lib/intel64/cc4.1.0_libc2.4_kernel2.6.16.21
```

```
export LIBRARY_PATH=$LIBRARY_PATH:/usr/local/\
tbb40_233oss/lib/intel64/cc4.1.0_libc2.4_kernel2.6.16.21
```

```
export TBB_EXAMPLES_X_NOSHMEM=1
```

the makefile

The `makefile` contains the following:

```
TBB_ROOT=/usr/local/tbb40_233oss
```

```
hello_task_group:
```

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

compiling and running

```
$ make hello_task_group
g++ -I/usr/local/tbb40_233oss/include \  
    -L/usr/local/tbb40_233oss/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
 - environment variables, compiling, and running
- 3 using the `parallel_for`
 - raising complex numbers to a large power
 - speeding up the computations with `parallel_for`

raising complex numbers to a large power

Consider the following problem:

Input: $n \in \mathbb{Z}_{>0}$, $d \in \mathbb{Z}_{>0}$, $\mathbf{x} \in \mathbb{C}^n$.

Output: $\mathbf{y} \in \mathbb{C}^n$, $y_k = x_k^d$, for $k = 1, 2, \dots, n$.

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 `complex` with the type `double` we declare

```
#include <complex>
```

```
using namespace std;
```

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

random complex doubles

```
#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

```
#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";
}
```

computing powers

```
void compute_powers ( int n, dcplx *x,  
                      dcplx *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, dcplx *x,  
                      dcplx *y, int d )  
{  
    for(int i=0; i < n; i++) // y[i] = pow(x[i],d);  
    {  
        // pow is too efficient  
        dcplx 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    0m17.456s
user    0m17.451s
sys     0m0.001s
```

the main program

```
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);
}
```

the main program continued

```
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
 - environment variables, compiling, and running
- 3 using the `parallel_for`
 - raising complex numbers to a large power
 - speeding up the computations with `parallel_for`

the speedup

```
$ time /tmp/powers_serial 1000 1000000 0
```

```
real    0m17.456s
user    0m17.451s
sys     0m0.001s
```

```
$ time /tmp/powers_tbb 1000 1000000 0
```

```
real    0m1.579s
user    0m18.540s
sys     0m0.010s
```

The speedup: $\frac{17.456}{1.579} = 11.055$ with 12 cores.

the class ComputePowers

```
class ComputePowers
{
    dcplx *const c; // numbers on input
    int d;          // degree
    dcplx *result; // output
public:
    ComputePowers(dcplx x[], int deg, dcplx 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)
        {
            dcplx z(1.0,0.0);
            for(int j=0; j < d; j++) z = z*c[i];
            result[i] = z;
        }
    }
};
```

tbb/blocked_range.h

```
#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`

```
#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:

```
task_scheduler_init init(task_scheduler_init::automatic);  
  
parallel_for(blocked_range<size_t>(0,dim),  
             ComputePowers(r,deg,s));
```

reference materials

- **Intel Threading Building Blocks. Tutorial.**
Available online via <http://www.intel.com>.
- Robert D. Blumofe and Charles E. Leiserson:
Scheduling Multithreaded Computations by Work-Stealing.
In the Proceedings of the 35th Annual IEEE Conference on Foundations of Computer Science (FoCS 1994), pages 356-368.
- Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams and Katherine A. Yelick: **The Landscape of Parallel Computing Research: A View from Berkeley.** Technical Report No. UCB/EECS-2006-183 EECS Department, University of California, Berkeley, December 18, 2006.

Summary + Exercises

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.

Homework will be collected on Friday 10 February at noon.

Bring your answers to the problems of lectures 9, 10, and 11 to class.

You may work in pairs. Please hand in only one copy per pair.