

Multithreaded Programming with pthreads, OpenMP, and Others

Mike Bailey

Oregon State University



Multithreaded Programming

Definitions:

- “Concurrent Programming”: Tasks can occur in any order
- “Parallel Programming”: Simultaneous execution of concurrent tasks on different processors or cores

When is it good to use Multithreading?

- Where specific operations can become blocked, waiting for something else to happen
- Where specific operations can be CPU-intensive
- Where specific operations must respond to asynchronous I/O, including the user interface (UI)
- Where specific operations have higher or lower priority than other operations
- Where performance can be gained by overlapping I/O
- To manage independent behaviors in interactive simulations
- When you want to accelerate a single program on multicore CPU chips



Thread Safety

Be sure code is “thread-safe” (i.e., don’t keep internal state between calls).

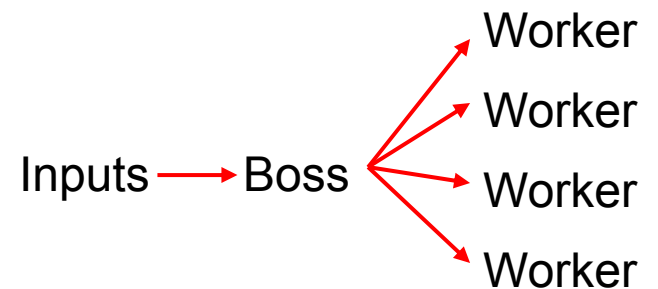
Note that many of the popular standard C routines (e.g., *strtok*) are not.

```
char *tok = strtok( Line, DELIMS );  
  
while( tok != NULL )  
{  
    ...  
    tok = strtok( NULL, DELIMS );  
};
```

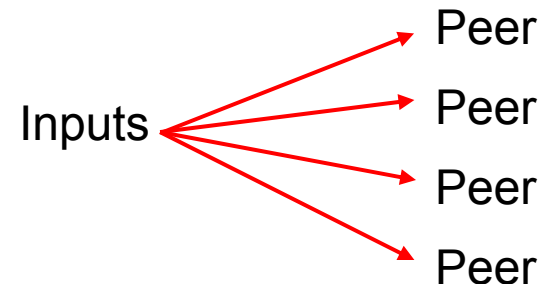
Multithreaded Programming Models

Boss / Worker I – all tasks come into a single “Boss Thread” who passes the tasks off to worker threads that it creates on the fly.

Boss / Worker II – all tasks come into a single “Boss Thread” who passes the tasks off to worker threads from a “thread pool” that the Boss created up front. (Avoids overhead of thread creation and destruction.)



Peer – specific-task threads are created up front, all tasks come into the correct thread



Pipeline – all tasks come into the first thread, who does a specific operation then passes them onto the second thread, etc.



Three Basic Types of Multithreading Programming Paradigms

1. You explicitly spawn off your own threads

You do all the work

You have very fine control

pthread and Java threads are good examples of this

2. You tell the compiler what it is allowed to multithread, and possibly how much

You do very little work – this is probably the best results-per-time

You have only coarse control

OpenMP is a good example of this

3. You specify what your tasks are and the API decides how to multithread it

Looks very much like #2

But with more control

Threading Building Blocks and Concurrent Collections are good examples of this



pthread Multithreaded Programming

- Pthreads is short for “Posix Threads”
- Posix is an IEEE standard for a Portable Operating System (section 1003.1c)
- Pthreads is a library you link with your program

The pthread paradigm is to let you spawn an application’s key procedures as separate threads

All threads share a single global heap (malloc, new)

Each thread has its own stack (procedure arguments, local variables)



Creating pthreads

The pthread paradigm is to spawn an application's key procedures as separate threads:

```
#include <pthread.h>

pthread_t      Thread1, Thread2;
int            *Subr1(), *Subr2();
int            Val1, Val2;

...

Val1 = 0;
int status1 = pthread_create( &Thread1, NULL, (void *) Subr1, (void *) &Val1 );
Val2 = 1;
int status2 = pthread_create( &Thread2, NULL, (void *) Subr2, (void *) &Val2 );

if( status1 == 0 )
    fprintf( stderr, "Thread 1 started successfully\n" );
else if( status1 == EAGAIN )
    fprintf( stderr, "Thread 1 failed because of insufficient resources\n" );
else if( status1 == EINVAL )
    fprintf( stderr, "Thread 1 failed because of invalid arguments\n" );
else
    fprintf( stderr, "Thread 1 failed for unknown reasons\n" );

...
```

The NULL in pthread_create indicates that this thread's Attribute Object is being defaulted

Waiting for pthreads to Finish

The pthread paradigm to waiting until the threads have exited and thus “rejoin” the thread that spawned them:

```
int *statusp1, *statusp2;
pthread_join( &Thread1, &statusp1 );
pthread_join( &Thread2, &statusp2 );

if( *statusp1 != 0 )
    fprintf( stderr, "Thread 1 exited with status %d\n", *statusp );

if( *statusp2 != 0 )
    fprintf( stderr, "Thread 2 exited with status %d\n", *statusp );
```

A thread's status is the integer value that the spawned-off procedure returned

Synchronizing pthreads

The pthread paradigm is to create a mutual exclusion (“mutex”) lock that only one thread can acquire at a time:

```
pthread_mutex_t    Sync;
...
pthread_mutex_init( &Sync, NULL );
...

pthread_mutex_lock( &Sync );
    << code that needs the mutual exclusion >>
pthread_mutex_unlock( &Sync );

pthread_mutex_trylock( &Sync );
```

The NULL in pthread_mutex_init indicates that this Mutex’s Attribute Object is being defaulted

pthread_mutex_lock blocks, waiting for the mutex lock to become available

pthread_mutex_trylock does not block – this is good if there is some more computing that could be done if the lock is not yet available

Letting a pthread Identify Itself

```
pthread_t self = pthread_self( );  
  
if( pthread_equal( self, io_thread ) )  
{  
    ...  
}
```

Canceling another pthread

```
pthread_cancel( Thread1 );
```



OpenMP Multithreaded Programming

- OpenMP is a multi-vendor standard

The OpenMP paradigm is to issue C/C++ “pragmas” to tell the compiler how to build the threads into the executable

```
#pragma omp directive [clause]
```

All threads share a single global heap (malloc, new)

Each thread has its own stack (procedure arguments, local variables)

OpenMP probably gives you the biggest multithread benefit per amount of work put in to using it



Creating OpenMP threads in Loops

```
#include <omp.h>
int i;

#pragma omp parallel for private(i)
for( i = 0; i < num; i++ )
{
    ...
}
```

This tells the compiler to parallelize the for loop into multiple threads, and to give each thread its own personal copy of the variable *i*. But, you don't have to do this for variables defined in the loop body:

```
#pragma omp parallel for
for( int i = 0; i < num; i++ )
{
    ...
}
```

Creating Sections of OpenMP Threads

```
#pragma omp parallel sections
{
    #pragma omp section
    {
        ...
    }
    #pragma omp section
    {
        ...
    }
}
```

This tells the compiler to place each section of code into its own thread

If each section contains a procedure call, then this is a good way to approximate the pthreads paradigm

Number of OpenMP threads

Two ways to specify how many OpenMP threads you want to have available:

1. Set the OMP_NUM_THREADS environment variable
2. Call `omp_set_num_threads(num);`

Asking how many cores this program has access to:

```
num = omp_get_num_procs( );
```

Setting the number of threads to the exact number of cores available:

```
num = omp_set_num_threads( omp_get_num_procs( ) );
```

Asking how many OpenMP threads this program is using:

```
num = omp_get_num_threads( );
```

Asking which thread this one is:

```
me = omp_get_thread_num( );
```



Data-Level Parallelism (DPL) in OpenMP

These last two calls are especially important if you want to do Data-Level Parallelism (DLP) !

```
total = omp_get_num_threads( );  
#pragma omp parallel private(me)  
    me = omp_get_thread_num( );  
    DoWork( me, total );  
#pragma omp end parallel
```

Enabling OpenMP in Visual Studio

1. To enable OpenMP in VS, go to the Project menu → Project Properties
2. Change the setting Configuration Properties → C/C++ → Language → OpenMP Support to "Yes (/openmp)"



More on Creating OpenMP threads in Loops

Normally, variables are shared among the threads. Each thread receives its own copy of private variables. Any temporary intermediate-computation variables defined outside the loop must be private:

```
float x, y;

#pragma omp parallel for private(x,y)
for( int i = 0; i < num; i++ )
{
}
```

Variables that accumulate are especially critical. They can't be private, but they also must be handled carefully so they don't get out of sync.

```
#pragma omp parallel for private(i,partialSum) reduction(+:total)
for( int i = 0; i < num; i++ )
{
    // compute a partial sum and add it to the total
    float partialSum = . . .
    total += partialSum;
}
```



Synchronizing OpenMP threads

The OpenMP paradigm is to create a mutual exclusion lock that only one thread can set at a time:

```
omp_lock_t Sync;  
...  
omp_init_lock( &Sync );  
...  
  
omp_set_lock( &Sync );  
    << code that needs the mutual exclusion >>  
omp_unset_lock( &Sync );  
  
omp_test_lock( &Sync );
```

`omp_set_lock` blocks, waiting for the lock to become available

`omp_test_lock` does not block – this is good if there is some more computing that could be done if the lock is not yet available



Other OpenMP Operations

See if OpenMP is available:

```
#ifdef _OPENMP  
...  
#endif
```

Force all threads to wait until all threads have reached this point:

```
#pragma omp barrier
```

(Note: there is an implied barrier after parallel for loops and OpenMP sections, unless the *nowait* clause is used)

Make this operation atomic (i.e., cannot be split by thread-swapping):

```
#pragma omp atomic  
x += 5.;
```

(Note: this is important for read-modify-write operations like this one)

Others: Intel's Threading Building Blocks (TBB)

- Developed by Intel
- Library-based (like pthreads)
- Especially emphasizes loop-parallelism (like OpenMP)
- Emphasizes task, not thread, management
- Expects you to have several times more tasks to do than threads to do them, to allow task-stealing to load-balance
- Heavily C++ template-ized

Others: Java

- Part of the standard Java release
- Initiate an object of type *Thread* and send it a *start()* message
- That causes it to spawn the contents of its *run()* method into a separate thread. When *run()* returns, the thread dies.

Others: C++0x

- Part of the extended standard of C++0x (in much the same way that the Standard Template Library is)
- Has *Thread* and *Mutex* classes



Others: NVIDIA's CUDA

- A C-like language to gain access to NVIDIA GPU cores for general-purpose computing
- “Compute Unified Device Architecture”
- Very much a data-parallel (SPMD) model, since it runs on a GPU
- You write one program, but designate a C/C++ part of it to run on the CPU and a CUDA part to run on multiple GPU cores
- Expects that you will create thousands of threads
- This is because GPUs have very little cache, so threads block often waiting for memory accesses to complete

Others: OpenCL

- A C-like language, originally proposed by Apple, now an industry standard
- Same application space as CUDA
- You write one program, but designate a C/C++ part of it to run on the CPU and an OpenCL part to run on the GPU
- No threads in the OpenCL part (but can use them in the C/C++ part)
- The OpenCL part of the code is vector-oriented, meaning that it can perform a single instruction on multiple data values at the same time (SIMD). Vector data types are: char_n , int_n , float_n , where $n = 2, 4, 8, \text{ or } 16$

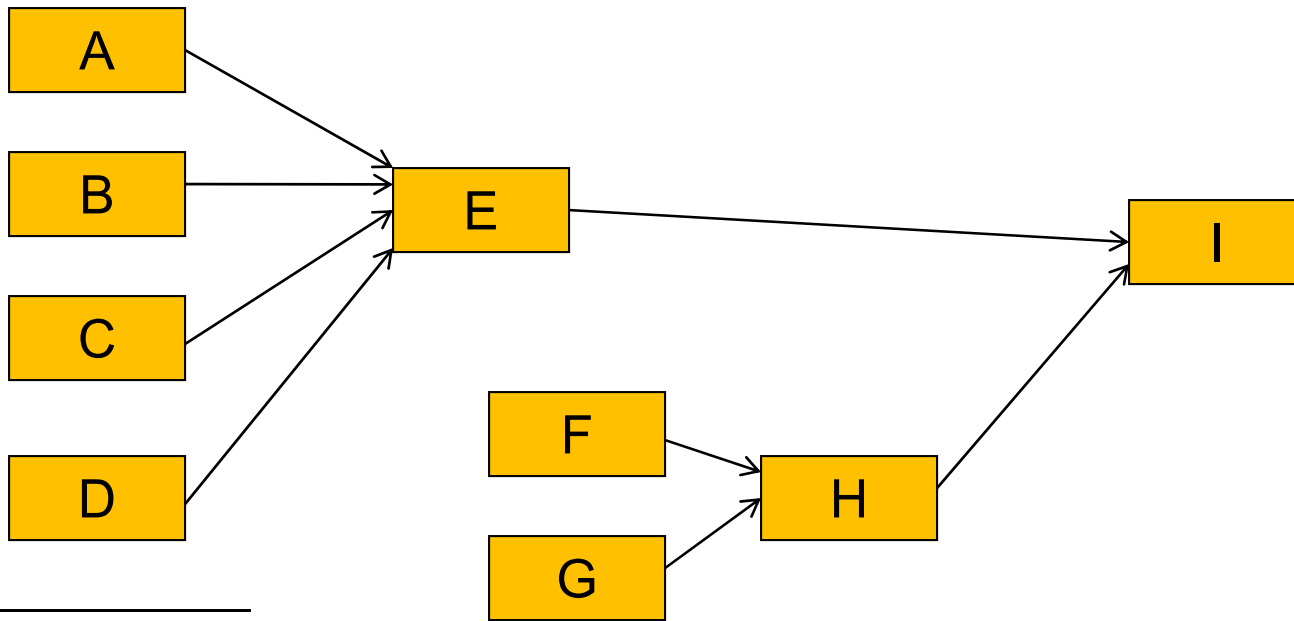
```
float4 f, g;
f = (float4)( 1.f, 2.f, 3.f, 4.f );
float16 a16, x16, y16, z16;

f.x = 0.;
f.xy = g.zw;
x16.s89ab = f;

float16 a16 = x16 * y16 + z16;
```

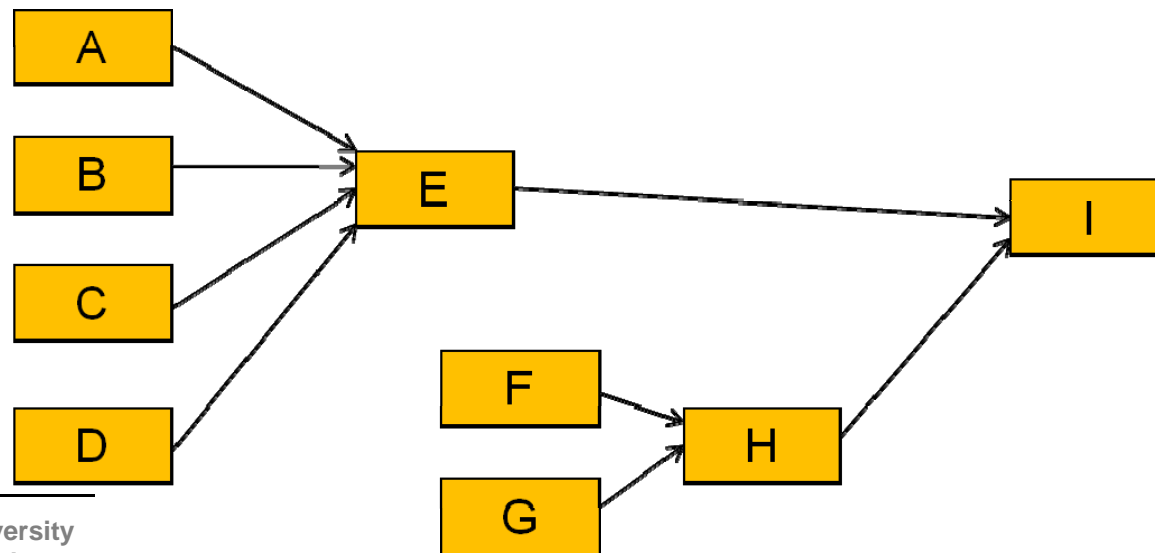
Others: Intel's Concurrent Collections for C/C++

- A library, targeted towards domain-specific (i.e., non-CS) development
- You arrange tasks in a dependency graph
- When a node's required inputs are all available, that node can run
- Can support DLP, TLP, or the pipeline pattern
- Nodes can have different priorities



Others: Intel's Larrabee

- Larrabee is a many-core x86-based chip.
- It's multitask library, called XNTask looks very much like Concurrent Collections
- You arrange tasks in a dependency graph
- When a node's required inputs are all available, that node can run
- Can support DLP, TLP, or the pipeline pattern
- Nodes can have different priorities



Some References

Maurice Herlihy and Nir Shavit, *The Art of Multiprocessor Programming*, Morgan Kaufmann, 2008.

Bradford Nichols, Dick Buttlar, and Jacqueline Proudex Farrell, *Pthreads Programming*, O'Reilly, 1998.

Rohit Chandra, Leonardo Dagun, Dave Kohr, Dror Maydan, Jeff McDonald, Ramesh Menon, *Parallel Programming in OpenMP*, Morgan Kaufmann, 2001..

James Reinders, *Intel Threading Building Blocks*, O'Reilly, 2007.

<http://software.intel.com/en-us/articles/pre-release-license-agreement-for-intel-concurrent-collections-for-cc-accept-end-user-license-agreement-and-download>

