“In theory, there is no difference between theory and practice. In practice there is.”
-- Yogi Berra
Before We Get Started…

- Last time
  - Wrapped up GPU computing w/ thrust
  - Wrapped up GPU computing discussion

- Today:
  - Parallel computing on the CPU
  - Get started with OpenMP for parallel computing on multicore CPUs

- Miscellaneous
  - HW07 posted online
    - Due on Oct. 28 at 11:59 PM
  - Due date for midterm project topic is tonight at 11:59 PM (upload in Learn@UW)
  - Exam moved back from November 8 to November 25 at 7:15 PM (Room TBA)
    - Review session held during regular class hour (show up only if you think it’s useful)
Quick Look at Hardware

- Intel Haswell
  - Released in June 2013
  - 22 nm technology
  - Transistor budget: 1.4 billions
    - Tri-gate, 3D transistors
  - Typically comes in four cores
  - Has an integrated GPU
  - Deep pipeline – 16 stages
  - Very strong machinery for ILP acceleration
  - Superscalar
  - Supports HTT (hyper-threading technology)

Good source of information for these slides: http://www.realworldtech.com/
Quick Look at Hardware

- Actual layout of the chip

![Image of chip layout]

- Schematic of the chip organization

- LLC: last level cache (L3)
- Three clocks:
  - A core’s clock ticks at 2.7 to 3.0 GHz but adjustable up to 3.7-3.9 GHz
  - Graphics processor ticking at 400 MHz but adjustable up to 1.3 GHz
  - Ring bus and the shared L3 cache - a frequency that is close to but not necessarily identical to that of the cores
Quick Look at Hardware

- System on Chip (SoC)
  - So many transistors, you can get creative…
  - The CPU integrates now functionality that used to reside mostly on the north bridge
  - Examples:
    - Voltage regulator
    - Display engine
    - Direct media interface (DMI) controller
    - PCI controller
    - Integrated memory controller (IMC)
  - Functional units to provide these services combine to form the “System Agent”
    - Used to be called the “uncore”
Caches

- Data:
  - L1 – 32 KB per core
  - L2 – 512 KB or 1024 KB per core
  - L3 – 8 MB per CPU

- Instruction:
  - L0 – room for about 1500 microoperations (uops) per core
    - See H/S primer, online
  - L1 – 32 KB per core

- Cache is a black hole for transistors
  - Example: 8 MB of L3 translates into:
    - $8 \times 1024 \times 1024 \times 8 \text{ (bits)} \times 6 \text{ (transistors per bit, SRAM)} = 402 \text{ million transistors out of 1.4 billions}$

- Caches are *very* important for good performance
Haswell Microarchitecture

[30,000 Feet]

- Microarchitecture components:
  - Instruction pre-fetch support (purple)
  - Instruction decoding support (orange)
    - CISC into uops
      - Turning CISC to RISC
  - Instruction Scheduling support (yellowish)
  - Instruction execution
    - Arithmetic (blue)
    - Memory related (green)

- More details: the primer posted online

[http://www.realworldtech.com]→
Moving from HW to SW
Acknowledgements

- Majority of slides used for discussing OpenMP issues are from Intel’s library of presentations for promoting OpenMP
  - Slides used herein with permission

- Credit given where due: IOMPP
  - IOMPP stands for “Intel OpenMP Presentation”
Data vs. Task Parallelism

- **Data parallelism**
  - You have a large amount of data elements and each data element (or possibly a subset of elements) needs to be processed to produce a result
  - When this processing can be done in parallel, we have data parallelism
  - Example:
    - Adding two long arrays of doubles to produce yet another array of doubles

- **Task parallelism**
  - You have a collection of tasks that need to be completed
  - If these tasks can be performed in parallel you are faced with a task parallel job
  - Examples:
    - Reading the newspaper, whistling, and scratching your back
    - The simultaneous breathing of your lungs, beating of your heart, liver function, controlling the swallowing, etc.
Objectives

- Understand OpenMP at the level where you can
  - Implement data parallelism
  - Implement task parallelism
- Provide an overview of OpenMP in three lectures
Work Plan

- What is OpenMP?
  - Parallel regions
  - Work sharing
  - Data environment
  - Synchronization

- Advanced topics
OpenMP: Target Hardware

- CUDA: targeted parallelism on the GPU

- OpenMP: targets parallelism on SMP architectures
  - Handy when
    - You have a machine that has 64 cores
    - You have a large amount of shared memory, say 128GB

- MPI: targeted parallelism on a cluster (distributed computing)
  - Note that MPI implementation can handle transparently an SMP architecture such as a workstation with two hexcore CPUs that draw on a good amount of shared memory
OpenMP: What’s Reasonable to Expect

- If you have 64 cores available to you, it is *highly* unlikely to get a speedup of more than 64 (superlinear)

- Recall the trick that helped the GPU hide latency
  - Overcommitting the SPs and hiding memory access latency with warp execution

- This mechanism of hiding latency by overcommitment does not *explicitly* exist for parallel computing under OpenMP beyond what’s offered by HTT
  - It exists implicitly, under the hood, through ILP support
OpenMP: What Is It?

- Portable, shared-memory threading API
  - Fortran, C, and C++
  - Multi-vendor support for both Linux and Windows

- Standardizes task & loop-level parallelism
- Supports coarse-grained parallelism
- Combines serial and parallel code in single source
- Standardizes ~ 20 years of compiler-directed threading experience

- Current spec is OpenMP 3.1
  - Released in October 2013
  - http://www.openmp.org
  - More than 300 Pages
**pthreads**: An OpenMP Precursor

- Before there was OpenMP, a common approach to support parallel programming was by use of *pthreads*
  - “pthread”: POSIX thread
  - POSIX: Portable Operating System Interface [for Unix]

- *pthreads*
  - Available originally under Unix and Linux
  - Windows ports are also available some as open source projects

- Parallel programming with *pthreads*: relatively cumbersome, prone to mistakes, hard to maintain/scale/expand
  - Not envisioned as a mechanism for writing scientific computing software
int main(int argc, char *argv[]) {
    parm *arg;
    pthread_t *threads;
    pthread_attr_t pthread_custom_attr;

    int n = atoi(argv[1]);

    threads = (pthread_t *) malloc(n * sizeof(*threads));
    pthread_attr_init(&pthread_custom_attr);

    barrier_init(&barrier1); /* setup barrier */
    finals = (double *) malloc(n * sizeof(double)); /* allocate space for final result */

    arg=(parm *)malloc(sizeof(parm)*n);
    for (int i = 0; i < n; i++) /* Spawn thread */
        { /* Spawn thread */
            arg[i].id = i;
            arg[i].noproc = n;
            pthread_create(&threads[i], & pthread_custom_attr, cpi, (void *)(arg+i));
        }

    for (int i = 0; i < n; i++) /* Synchronize the completion of each thread. */
        pthread_join(threads[i], NULL);

    free(arg);
    return 0;
}
```c
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <sys/types.h>
#include <pthread.h>
#include <sys/time.h>

#define SOLARIS 1
#define ORIGIN  2
#define OS SOLARIS

typedef struct {
    int   id;
    int   noproc;
    int   dim;
} parm;

typedef struct {
    int   cur_count;
    pthread_mutex_t barrier_mutex;
    pthread_cond_t barrier_cond;
} barrier_t;

void barrier_init(barrier_t * mybarrier) {
    /* must run before spawning the thread */
    pthread_mutexattr_t attr;

    # if (OS==ORIGIN)
        pthread_mutexattr_setprotocol(&attr, PTHREAD_PRIO_INHERIT);
        pthread_mutexattr_setscope(&attr, 0);
        pthread_mutex_init(&(mybarrier->barrier_mutex), &attr);
    # elif (OS==SOLARIS)
        pthread_mutex_init(&(mybarrier->barrier_mutex), NULL);
    # else
        # error "undefined OS"
    # endif

    pthread_cond_init(&(mybarrier->barrier_cond), NULL);
    mybarrier->cur_count = 0;
}

void barrier(int numproc, barrier_t * mybarrier) {
    pthread_mutex_lock(&(mybarrier->barrier_mutex));
    mybarrier->cur_count++;
    if (mybarrier->cur_count!=numproc) {
        pthread_cond_wait(&(mybarrier->barrier_cond), &(mybarrier->barrier_mutex));
    } else {
        mybarrier->cur_count=0;
        pthread_cond_broadcast(&(mybarrier->barrier_cond));
    }
    pthread_mutex_unlock(&(mybarrier->barrier_mutex));
}

void* cpi(void *arg) {
    parm  *p = (parm *) arg;
    int     myid = p->id;
    int     numprocs = p->noproc;
    double  PI25DT = 3.141592653589793238462643;
    double  mypi, pi, h, sum, x, a;
    double  startwtime, endwtime;

    if (myid == 0) {
        startwtime = clock();
    }
    barrier(numprocs, &barrier1);
    if (rootn==0)
        finals[myid]=0;
    else {
        h = 1.0 / (double) rootn;
        sum = 0.0;
        for(int i = myid + 1; i <= rootn; i += numprocs) {
            x = h * ((double) i - 0.5);
            sum += f(x);
        }
        mypi = h * sum;
    }
    finals[myid] = mypi;
    barrier(numprocs, &barrier1);

    if (myid == 0){
        pi = 0.0;
        for(int i=0; i < numprocs; i++) pi += finals[i];
        endtime = clock();
        printf("pi is approx %.16f, Error is %.16f\n", pi, fabs(pi - PI25DT));
        printf("wall clock time = %f\n", (endtime - startwtime) / CLOCKS_PER_SEC);
    }
    return NULL;
}
```
Looking at the previous example (which is not the best written piece of code, lifted from the web…)

- Code displays platform dependency (not portable)
- Code is cryptic, low level, hard to read and maintain
- Requires busy work: fork and joining threads, etc.
  - Burdens the developer
  - Probably in the way of the compiler as well: rather low chances that the compiler will be able to optimize the implementation

Higher level approach to SMP parallel computing for *scientific applications* was in order
OpenMP Programming Model

- **Master thread** spawns a **team of threads** as needed
  - Managed transparently on your behalf
  - It still relies on thread fork/join methodology to implement parallelism
    - The developer is spared the details

- Parallelism is added incrementally: that is, the sequential program evolves into a parallel program
OpenMP: Library Support

- Runtime environment routines:
  - Modify/check the number of threads
    
    \[ \text{omp}\_[\text{set|get}]\_\text{num\_threads}() \]
    
    \[ \text{omp}\_\text{get\_thread\_num}() \]
    
    \[ \text{omp}\_\text{get\_max\_threads}() \]

- Are we in a parallel region?
  
  \[ \text{omp\_in\_parallel}() \]

- How many processors in the system?
  
  \[ \text{omp}\_\text{get\_num\_procs}() \]

- Explicit locks
  
  \[ \text{omp}\_[\text{set|unset}]\_\text{lock}() \]

- And several more...
  
  [https://computing.llnl.gov/tutorials/openMP/](https://computing.llnl.gov/tutorials/openMP/)
A Few Syntax Details to Get Started

- Picking up the API - header file in C, or Fortran 90 module
  
  ```
  #include "omp.h"
  use omp_lib
  ```

- Most of the constructs in OpenMP are compiler directives or pragmas
  
  - For C and C++, the pragmas take the form:
    
    ```
    #pragma omp construct [clause [clause]...]
    ```

  - For Fortran, the directives take one of the forms:
    
    ```
    C$OMP construct [clause [clause]...]
    !$OMP construct [clause [clause]...]
    !$OMP construct [clause [clause]...]
    ```
Why Compiler Directive and/or Pragmas?

- One of OpenMP’s design principles was to have the same code, with no modifications and have it run either on an one core machine, or a multiple core machine
- Therefore, you have to “hide” all the compiler directives behind Comments and/or Pragmas
- These hidden directives would be picked up by the compiler only if you instruct it to compile in OpenMP mode
  - Example: Visual Studio – you have to have the /openmp flag on in order to compile OpenMP code
  - Also need to indicate that you want to use the OpenMP API by having the right header included: #include <omp.h>

**Step 1:** Go here

**Step 2:** Select /openmp
OpenMP, Compiling Using the Command Line

- Method depends on compiler

- GCC:
  
  ```
  $ gcc -o integrate_omp integrate_omp.c -fopenmp
  ```

- ICC:
  
  ```
  $ icc -o integrate_omp integrate_omp.c -openmp
  ```

- MSVC (not in the express edition):
  
  ```
  $ cl /openmp integrate_omp.c
  ```
Enabling OpenMP with CMake

```cmake
# Minimum version of CMake required. cmake_minimum_required(VERSION 2.8)

# Set the name of your project project(ME964-omp)

# Include macros from the SBEL util library include(SBELUtils.cmake)

# Example OpenMP program enable_openmp_support()
add_executable(integrate_omp integrate_omp.cpp)

find_package("OpenMP" REQUIRED)
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} ${OpenMP_C_FLAGS}"
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${OpenMP_CXX_FLAGS}"))
```

With the template

Without the template

Replaces include(SBELUtils.cmake) and enable_openmp_support() above
OpenMP Odds and Ends…

- Controlling the number of threads
  - The default number of threads that a program uses when it runs is the number of online processors on the machine

- For the C Shell: setenv OMP_NUM_THREADS number

- For the Bash Shell: export OMP_NUM_THREADS=number

- Timing:

```
#include <omp.h>
stime = omp_get_wtime();
mylongfunction();
etime = omp_get_wtime();
total=etime-stime;
```
Work Plan

● What is OpenMP?
  Parallel regions
  Work sharing
  Data environment
  Synchronization

● Advanced topics
Parallel Region & Structured Blocks (C/C++)

- Most OpenMP constructs apply to structured blocks
  - Structured block, definition: a block with one point of entry at the top and one point of exit at the bottom
  - The only “branches” allowed are exit() function calls in C/C++

A structured block
```c
#pragma omp parallel
{
    int id = omp_get_thread_num();
    more: res[id] = do_big_job (id);
    if (not_conv (res[id]) goto more;
}
printf ("All done\n");
```

Not a structured block
```c
if (go_now()) goto more;
#pragma omp parallel
{
    int id = omp_get_thread_num();
    more: res[id] = do_big_job(id);
    if (conv (res[id]) goto done;
        goto more;
}
done:
    if (!really_done()) goto more;
```

There is an implicit barrier at the right “}” curly brace and that’s the point at which the other worker threads complete execution and either go to sleep or spin or otherwise idle.
Example: Hello World on my Machine

```c
#include <stdio.h>
#include <omp.h>

int main() {
  #pragma omp parallel
  {
    int myId = omp_get_thread_num();
    int nThreads = omp_get_num_threads();

    printf("Hello World. I'm thread %d out of %d.\n", myId, nThreads);
    for (int i=0; i<2 ;i++)
      printf("Iter:%d\n",i);
  }
  printf("GoodBye World\n");
}
```

- Here’s my machine (12 core machine):
  Two Intel Xeon X5650 Westmere 2.66GHz 12MB L3 Cache LGA 1366 95Watts Six-Core Processors
OpenMP: Important Remark

- One of the key tenets of OpenMP is that of **data independence** across parallel jobs.

- Specifically, when distributing work among parallel threads it is assumed that there is no data dependency.

- Since you place the `omp parallel` directive around some code, it is **your responsibility** to make sure that data dependency is ruled out:
  - Compilers are not smart enough and sometimes and they can’t identify data dependency between what might look as independent parallel jobs.
Work Plan

- What is OpenMP?
  - Parallel regions
  - **Work sharing**
  - Data environment
  - Synchronization

- Advanced topics
Work Sharing

- **Work sharing** is the general term used in OpenMP to describe distribution of work across threads.

- Three primary avenues for work sharing in OpenMP:
  - “omp for” construct
  - “omp sections” construct
  - “omp task” construct

Each of them automatically divides work among threads.
“omp for” construct

```c
// assume N=12
#pragma omp parallel
#pragma omp for
  for(i = 1, i < N+1, i++)
    c[i] = a[i] + b[i];
```

- Threads are assigned an independent set of iterations
- Threads must wait at the end of work-sharing construct
Combining Constructs

- These two code segments are equivalent

```c
#pragma omp parallel
{
    #pragma omp for
    for (int i=0; i< MAX; i++) {
        res[i] = huge();
    }
}
```

```c
#pragma omp parallel for
for (int i=0; i< MAX; i++) {
    res[i] = huge();
}
```
The Private Clause

- Reproduces the variable for each task
  - By declaring a variable as being private it means that each thread will have a private copy of that variable
    - The value that thread 1 stores in x is different than the value that thread 2 stores in the variable x
  - Variables are un-initialized; C++ object is default constructed

```c
void* work(float* c, int N) {
    float x, y;
    int i;
    #pragma omp parallel for private(x,y)
    for(i=0; i<N; i++) {
        x = a[i]; y = b[i];
        c[i] = x + y;
    }
}
```
Example: Parallel Mandelbrot

- Objective: create a parallel version of Mandelbrot using OpenMP work sharing clauses to parallelize the computation of Mandelbrot.
Example: Parallel Mandelbrot
[The Important Function; Includes material from IOMPP]

```c
int Mandelbrot (float z_r[][JMAX], float z_i[][JMAX], float z_color[][JMAX], char gAxis){
    float xinc = (float)XDELTA/(IMAX-1);
    float yinc = (float)YDELTA/(JMAX-1);

    #pragma omp parallel for private(i,j) schedule(static,8)
    for (int i=0; i<IMAX; i++) {
        for (int j=0; j<JMAX; j++) {
            z_r[i][j] = (float) -1.0*XDELTA/2.0 + xinc * i;
            z_i[i][j] = (float) 1.0*YDELTA/2.0 - yinc * j;
            switch (gAxis) {
                case 'V':
                    z_color[i][j] = CalcMandelbrot(z_r[i][j], z_i[i][j] ) /1.0001;
                    break;
                case 'H':
                    z_color[i][j] = CalcMandelbrot(z_i[i][j], z_r[i][j] ) /1.0001;
                default:
                    break;
            }
        }
    }

    return 1;
}
```
The **schedule** Clause

- The **schedule** clause affects how loop iterations are mapped onto threads

**schedule(static [,chunk])**
- Blocks of iterations of size “chunk” assigned to each thread
- Round robin distribution
- Low overhead, may cause load imbalance

**schedule(dynamic[,chunk])**
- Threads grab “chunk” iterations
- When done with iterations, thread requests next set
- Higher threading overhead, can reduce load imbalance

**schedule(guided[,chunk])**
- Dynamic schedule starting with large block
- Size of the blocks shrink; no smaller than “chunk”
schedule Clause Example

```c
#pragma omp parallel for schedule (static, 8)
    for( int i = start; i <= end; i += 2 )
    {
        if ( TestForPrime(i) )  gPrimesFound++;
    }
```

- Iterations are divided into chunks of 8
- If start = 3, then first chunk is

\[ i=\{3,5,7,9,11,13,15,17\} \]
Work Plan

- What is OpenMP?
  Parallel regions
  Work sharing – Parallel Sections
  Data environment
  Synchronization

- Advanced topics