“The competent programmer is fully aware of the strictly limited size of his own skull; therefore he approaches the programming task in full humility, and among other things he avoids clever tricks like the plague.”

Edsger W. Dijkstra
Before We Get Started…

- Last time
  - Learn how to run an MPI executable on Newton
  - Point-to-Point Communication with MPI
  - Collective Communication in MPI

- Today
  - Parallel Computing using OpenMP, part 1 of 2.

- Other issues
  - Assignment 7 was posted on the class website, due on April 7
  - Class website includes link to the OpenMP 3.0 Application Programming Interface
The overwhelming majority of slides used for discussing OpenMP issues are from Intel’s library of presentations for promoting OpenMP.
- The slides are used herein with permission.

Credit is given where due by a “Credit: IOMPP” or “Includes material from IOMPP” message at the bottom of the slide.
- 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, drinking coffee, and scratching your back
    - The breathing 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
Work Plan

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

- Advanced topics
OpenMP: Target Hardware

- CUDA: targeted parallelism on the GPU

- MPI: targeted parallelism on a cluster (distributed computing)
  - Note that MPI implementation can handle transparently a SMP architecture such as a workstation with two hexcore CPUs that use a large amount of shared memory

- OpenMP: targets parallelism on SMP architectures
  - Handy when
    - You have a machine that has 12 cores, probably 24 if HTT is accounted for
    - You have a large amount of shared memory that is backed by a 64 bit OS
OpenMP: What to Expect

- If you have 12 cores available to you, it is *highly* unlikely to get a speedup of more than 12 (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
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 3.0
  - [http://www.openmp.org](http://www.openmp.org)
  - 318 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
  - Moreover, not envisioned as a mechanism for writing scientific computing software
“pthreads”: Example

```c
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 */
        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) {
  /* barrier */
  pthread_mutexattr_t attr;
  # if (OS==ORIGIN)
    pthread_mutexattr_setprotocol(&attr, PTHREAD_PRIO_INHERIT);
    pthread_mutexattr_setprioceiling(&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_mutex_init(&(mybarrier->barrier_mutex), 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;
}
```

“pthreads”: Moving Away…

- 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 (not simple)
  - 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

- Long time experience with “pthreads” suggested that a 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

Includes material from IOMPP
OpenMP: 20+ Library Routines

- Runtime environment routines:
  - Modify/check the number of threads
    - `omp_[set|get]_num_threads()`
    - `omp_get_thread_num()`
    - `omp_get_max_threads()`
  - Are we in a parallel region?
    - `omp_in_parallel()`
  - How many processors in the system?
    - `omp_get_num_procs()`
  - Explicit locks
    - `omp_[set|unset]_lock()`
  - And several more...
A Few Syntax Details to Get Started

- 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]...]
    ```

- Header file or Fortran 90 module
  ```
  #include "omp.h"
  use omp_lib
  ```

Credit: IOMPP
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 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
Work Plan

- What is OpenMP?
  - Parallel regions
  - Work sharing
  - Data environment
  - Synchronization
- Advanced topics
Most OpenMP constructs apply to structured blocks

- Structured block: a block with one point of entry at the top and one point of exit at the bottom
- The only “branches” allowed are STOP statements in Fortran and exit() in C/C++

```
#pragma omp parallel
{
    int id = omp_get_thread_num();

    more: res[id] = do_big_job(id);

    if (conv (res[id])) goto more;
}
printf("All done\n");
```

```
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;
```

A structured block

Not a structured block
```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");
}
```

Example: Hello World on my Machine

Here’s my machine (12 core machine)

Two Intel Xeon X5650 Westmere 2.66GHz
12MB L3 Cache LGA 1366 95Watts Six-Core Processors

Credit: OpenMP code from IOMPP
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 it is outright impossible to rule out data dependency between what might look as independent parallel jobs.
Work Plan

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

● Advanced topics
Work Sharing

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

- Three categories of worksharing in OpenMP:
  - “omp for” construct
  - “omp sections” construct
  - “omp task” construct

Automatically divides work among threads

Credit: IOMPP
“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
  - Variables are un-initialized; C++ object is default constructed
  - Any value external to the parallel region is undefined
  - 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

```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.

Includes material from IOMPP
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” to threads
  - 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

```
#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
Function Level Parallelism

```c
a = alice();
b = bob();
s = boss(a, b);
c = cy();
printf("%6.2f\n", bigboss(s,c));
```

alice, bob, and cy can be computed in parallel
#pragma omp sections
Must be inside a parallel region
Precedes a code block containing $N$ sub-blocks of code that may be executed concurrently by $N$ threads
Encompasses each omp section

#pragma omp section
Precedes each sub-block of code within the encompassing block described above
Enclosed program segments are distributed for parallel execution among available threads
#pragma omp parallel sections
{
    #pragma omp section
        double a = alice();
    #pragma omp section
        double b = bob();
    #pragma omp section
        double c = cy();
}

    double s = boss(a, b);
    printf ("%6.2f\n", bigboss(s,c));
Advantage of Parallel Sections

- Independent sections of code can execute concurrently – reduce execution time

```c
#pragma omp parallel sections
{
#pragma omp section
  phase1();
#pragma omp section
  phase2();
#pragma omp section
  phase3();
}
```

Credit: IOMPP
Work Plan

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

- Advanced topics
New Addition to OpenMP

- **Tasks** – Main change for in the latest 3.0 version of OpenMP

- Allows parallelization of irregular problems
  - Unbounded loops
  - Recursive algorithms
  - Producer/consumer
Tasks: What Are They?

- Tasks are independent units of work
- A thread is assigned to perform a task
- Tasks might be executed immediately or might be deferred
  - The runtime system decides which of the above
- Tasks are composed of
  - **code** to execute
  - **data** environment
  - **internal control variables** (ICV)
Simple Task Example

A pool of 8 threads is created here

Only one thread gets to execute the while loop

The single “while loop” thread creates a task for each instance of processwork()
Task Construct – Explicit Task View

- A team of threads is created at the **omp parallel** construct

- A single thread is chosen to execute the while loop – call this thread “L”

- Thread L operates the while loop, creates tasks, and fetches next pointers

- Each time L crosses the **omp task** construct it generates a new task and has a thread assigned to it

- Each task runs in its own thread

- All tasks complete at the barrier at the end of the parallel region’s construct

```c
#pragma omp parallel
{
    #pragma omp single
    {
        // block 1
        node *p = head_of_list;
        while (p) {
            //block 2
            #pragma omp task private(p)
            process(p);
            p = p->next;  //block 3
        }
    }
}
```

Credit: IOMPP
Why are tasks useful?

Have potential to parallelize irregular patterns and recursive function calls

```c
#pragma omp parallel
{
    #pragma omp single
    { // block 1
        node *p = head_of_list;
        while (p) { //block 2
            #pragma omp task private(p)
            process(p);
            p = p->next; //block 3
        }
    }
}
```

Includes material from IOMPP
Tasks: Synchronization Issues

- **Setup:**
  - Assume Task B specifically relies on completion of Task A
  - You need to be in a position to guaranteed completion of Task A before invoking the execution of Task B

- Tasks are guaranteed to be complete at thread or task barriers:
  - At the directive: `#pragma omp barrier`
  - At the directive: `#pragma omp taskwait`
Task Completion Example

```c
#pragma omp parallel
{
    #pragma omp task
    foo();
    #pragma omp barrier
    #pragma omp single
    {
        #pragma omp task
        bar();
    }
}
```

- Multiple foo tasks created here – one for each thread
- All foo tasks guaranteed to be completed here
- One bar task created here
- bar task guaranteed to be completed here