Acknowledgements

- A large number of slides used for discussing OpenMP are from Intel’s library of presentations 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, drinking coffee, and scratching your back
    - The breathing your lungs, beating of your heart, liver function, controlling swallowing, etc.
Objectives

- Understand OpenMP at the level where you can
  - Implement data parallelism
  - Implement task parallelism
- Not able to provide a full overview of OpenMP
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 draw on a good amount of shared memory
- **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
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
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 4.0
  - http://www.openmp.org
  - More than 300 pages
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
```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 */
    /* must run before spawning the thread */
    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
    mybarrier->cur_count = 0;
}

void barriere_init(barrier_t * mybarrier) { /* barrier */
    /* must run before spawning the thread */
    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
    mybarrier->cur_count = 0;
}

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];
        endwtime = clock();
        printf("pi is approx %.16f, Error is %.16f\n", pi, fabs(pi - PI25DT));
        printf("wall clock time = %f\n", (endwtime - startwtime) / CLOCKS_PER_SEC);
    }
    return NULL;
}
```
ptthreads: leaving them behind…

- 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

- 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

![Diagram showing Master Thread and Parallel Regions]
OpenMP: Library Support

- Runtime environment routines:
  - Modify/check/get info about the number of threads
    - `omp_[set|get]_num_threads()`
    - `omp_get_thread_num();` // tells which thread you are
    - `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...

https://computing.llnl.gov/tutorials/openMP/
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`
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>.
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
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

- **G++:**
  
  ```
  $ g++ -o integrate_omp integrate_omp.c -fopenmp
  ```

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

- **Microsoft Visual Studio:**
  
  ```
  $ cl /openmp integrate_omp.c
  ```
Enabling OpenMP with 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 utils 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}"
OpenMP Odds and Ends…

- **Controlling the number of threads at runtime**
  - 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:**
  ```c
  #include <omp.h>
  stime = omp_get_wtime();
  longfunction();
  etime = omp_get_wtime();
  total=etime-stime;
  ```
OpenMP Odds and Ends…

- Controlling the number of threads at runtime
  - 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_NUM_THREADS number`
  - For the Bash Shell: `export OMP_NUM_THREADS=number`

- Timing:

  ```c
  #include <omp.h>
  stime = omp_get_wtime();
  longfunction();
  etime = omp_get_wtime();
  total=etime-stime;
  ```

  Use this on Euler
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: 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;
```
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 `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.
```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
One of the key tenets of OpenMP is the assumption 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
  - 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 work sharing in OpenMP:
  - “omp for” construct
  - “omp sections” construct
  - “omp task” construct
Work Sharing

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

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

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

// 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 variable 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

```cpp
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

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

- 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.
omp sections

- `#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
omp sections

- 
  #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
omp sections

- `#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
Functional Level Parallelism Using omp sections

```
#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));
```
Advanced of Parallel Sections

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

```
#pragma omp parallel sections
{
 #pragma omp section
   phase1();
 #pragma omp section
   phase2();
 #pragma omp section
   phase3();
}
```
#include <stdio.h>
#include <omp.h>

int main() {
    printf("Start with 2 procs\n");
#ifndef __omp_par
#pragma omp parallel sections num_threads(2)
#endif

#pragma omp section
{
    printf("Start work 1\n");
    double startTime = omp_get_wtime();
    while( (omp_get_wtime() - startTime) < 2.0);
    printf("Finish work 1\n");
}

#pragma omp section
{
    printf("Start work 2\n");
    double startTime = omp_get_wtime();
    while( (omp_get_wtime() - startTime) < 2.0);
    printf("Finish work 2\n");
}

#pragma omp section
{
    printf("Start work 3\n");
    double startTime = omp_get_wtime();
    while( (omp_get_wtime() - startTime) < 2.0);
    printf("Finish work 3\n");
}

return 0;
}
sections, Example: 2 threads
```c
#include <stdio.h>
#include <omp.h>

int main() {
    printf("Start with 4 procs\n");
    #pragma omp parallel sections num_threads(4)
    {
        #pragma omp section
        {
            printf("Start work 1\n");
            double startTime = omp_get_wtime();
            while( (omp_get_wtime() - startTime) < 2.0);
            printf("Finish work 1\n");
        }
        #pragma omp section
        {
            printf("Start work 2\n");
            double startTime = omp_get_wtime();
            while( (omp_get_wtime() - startTime) < 6.0);
            printf("Finish work 2\n");
        }
        #pragma omp section
        {
            printf("Start work 3\n");
            double startTime = omp_get_wtime();
            while( (omp_get_wtime() - startTime) < 2.0);
            printf("Finish work 3\n");
        }
    }
    return 0;
}
```
```c
#include <stdio.h>
#include <omp.h>

int main() {
    printf("Start with 4 procs\n");
    #pragma omp parallel sections num_threads(4)
    {
        #pragma omp section
        {
            printf("Start work 1\n");
            double startTime = omp_get_wtime();
            while( (omp_get_wtime() - startTime) < 2.0);
            printf("Finish work 1\n");
        }
        #pragma omp section
        {
            printf("Start work 2\n");
            double startTime = omp_get_wtime();
            while( (omp_get_wtime() - startTime) < 6.0);
            printf("Finish work 2\n");
        }
        #pragma omp section
        {
            printf("Start work 3\n");
            double startTime = omp_get_wtime();
            while( (omp_get_wtime() - startTime) < 2.0);
            printf("Finish work 3\n");
        }
    }
    return 0;
}
```
sections, Example: 4 threads
Work Plan

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

- Advanced topics
OpenMP Tasks

- **Task** – Most important feature added in 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)

Serial

Parallel

Time

[IOMPP]→
Tasks: What Are They?

[More specifics…]

- Code to execute
  - The literal code in your program enclosed by the task directive

- Data environment
  - The shared & private data manipulated by the task

- Internal control variables
  - Thread scheduling and environment variable type controls

- A task is a specific instance of executable code and its data environment, generated when a thread encounters a task construct

- Two activities: packaging and execution
  - A thread packages new instances of a task (code and data)
  - Some thread in the team executes the task at some later time
using namespace std;

typedef list<double> LISTDBL;

toDoSomething(LISTDBL::iterator& itrtr) {
    *itrtr *= 2.0;
}

int main() {
    LISTDBL test; // default constructor
    LISTDBL::iterator it;

    for (int i=0; i<4; ++i)
        for (int j=0; j<8; ++j) test.insert(test.end(), pow(10.0, i+1)+j);
    for (it = test.begin(); it!= test.end(); it++) cout << *it << endl;

    it = test.begin();
    #pragma omp parallel num_threads(8)
    {
        #pragma omp single private (it)
        {
            while (it != test.end()) {
                #pragma omp task
                {
                    doSomething(it);
                }
                it++;
            }
        }
    }
    for (it = test.begin(); it != test.end(); it++) cout << *it << endl;
    return 0;
}
Compile like:

```
$ g++ -o testOMP.exe testOMP.cpp
```
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
- Each task has its own stack space that will be destroyed when the task is completed

```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
    }
  }
}
```

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
        }
    }
}
```
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
        }
    }
}
```
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
        }
    }
}
```
Tasks: Synchronization Issues

- Setup:
  - Assume Task B specifically relies on completion of Task A
  - You need to be in a position to guarantee 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`
#pragma omp parallel
{
  #pragma omp task
  foo();
  #pragma omp barrier
  #pragma omp single
  {
    #pragma omp task
    bar();
  }
}
Task Completion Example

Multiple foo tasks created here – one for each thread

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

- What is OpenMP?
  - Parallel regions
  - Work sharing
  - Data scoping
  - Synchronization
- Advanced topics
Data Scoping – What’s shared

- OpenMP runs on SMP architectures → data lives in shared-memory

Nomenclature:
- OpenMP shared variable - variable that can be read/written by multiple threads

Shared clause can be used to make items explicitly shared
- Global variables are shared by default among tasks
- Other examples of variables being shared among threads
  - File scope variables
  - Namespace scope variables
  - Variables with const-qualified type having no mutable member
  - Static variables which are declared in a scope inside the construct
Data Scoping – What’s Private

- Not everything is shared...

  - Examples of implicitly determined PRIVATE variables:
    - Stack (local) variables in functions called from parallel regions
    - Automatic variables within a statement block
    - Loop iteration variables
    - Implicitly declared private variables within tasks will be treated as firstprivate

- **firstprivate**
  - Specifies that each thread should have its own instance of a variable, and that the variable should be initialized with the value of the variable, because it exists before the parallel construct
Data Scoping – The Golden Rule

- When in doubt, explicitly indicate who’s what
#pragma omp parallel shared(a,b,c,d,nthreads) private(i,tid)
{
    tid = omp_get_thread_num();
    if (tid == 0) {
        nthreads = omp_get_num_threads();
        printf("Number of threads = %d\n", nthreads);
    }
    printf("Thread %d starting...\n",tid);

    #pragma omp sections nowait
    {
        #pragma omp section
        {
            printf("Thread %d doing section 1\n",tid);
            for (i=0; i<N; i++)
            {
                c[i] = a[i] + b[i];
                printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
            }
        }

        #pragma omp section
        {
            printf("Thread %d doing section 2\n",tid);
            for (i=0; i<N; i++)
            {
                d[i] = a[i] * b[i];
                printf("Thread %d: d[%d]= %f\n",tid,i,d[i]);
            }
        }
    }  /* end of sections */

    printf("Thread %d done.\n",tid);
}  /* end of parallel section */

When in doubt, explicitly indicate who’s what
A Data Environment Example

float A[10];
main () {
    int index[10];
    #pragma omp parallel
    {
        Work (index);
    }
    printf("%d\n", index[1]);
}

extern float A[10];
void Work (int *index)
{
    float temp[10];
    static integer count;
    <...>
}

Assumed to be in another translation unit

Includes material from IOMPP
A Data Environment Example

Which variables are shared and which variables are private?

```c
float A[10];
main () {
    int index[10];
    #pragma omp parallel
    {
        Work (index);
    }
    printf ("%d\n", index[1]);
}

extern float A[10];
void Work (int *index)
{
    float temp[10];
    static integer count;
    ...
}
```

Assumed to be in another translation unit

Includes material from IOMPP
A, index, and count are shared by all threads, but temp is local to each thread.

```c
float A[10];
main () {
    int index[10];
    #pragma omp parallel
    {
        Work (index);
    }
    printf ("%d\n", index[1]);
}

extern float A[10];
void Work (int *index) {
    float temp[10];
    static integer count;
    ...
}
```

Assumed to be in another translation unit.

Includes material from IOMPP
Data Scoping Issue: fib Example

```c
int fib ( int n ) {
    int x, y;
    if ( n < 2 ) return n;
    #pragma omp task
    x = fib(n-1);
    #pragma omp task
    y = fib(n-2);
    #pragma omp taskwait
    return x+y;
}
```

n is private in both tasks

x is a private variable
y is a private variable

This is very important here

What’s wrong here?

Credit: IOMPP
Data Scoping Issue: fib Example

```c
int fib ( int n ) {
    int x, y;
    if ( n < 2 ) return n;
    #pragma omp task
    x = fib(n-1);
    #pragma omp task
    y = fib(n-2);
    #pragma omp taskwait
    return x+y;
}
```

Values of the private variables

- **n** is private in both tasks
- **x** is a private variable
- **y** is a private variable
- This is very important here
- What’s wrong here?

Credit: IOMPP
int fib ( int n ) {
    int x, y;
    if ( n < 2 ) return n;

    #pragma omp task
    {
        x = fib(n-1);
    }

    #pragma omp task
    {
        y = fib(n-2);
    }

    #pragma omp taskwait

    return x+y
}
```c
int fib ( int n ) {
    int x, y;
    if ( n < 2 ) return n;
    #pragma omp task shared(x)
    x = fib(n-1);
    #pragma omp task shared(y)
    y = fib(n-2);
    #pragma omp taskwait
    return x+y;
}
```

The values of the x & y variables will be available outside each task construct – after the taskwait.
Work Plan

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

Synchronization

- Advanced topics
Implicit Barriers

- Several OpenMP constructs have implicit barriers
  - parallel – necessary barrier – cannot be removed
  - for
  - single

- Unnecessary barriers hurt performance and can be removed with the `nowait` clause
  - The `nowait` clause is applicable to:
    - for clause
    - single clause
Nowait Clause

- Use when threads unnecessarily wait between independent computations

```c
#pragma omp for nowait
for(...)
{ [...];

#pragma omp for schedule(dynamic,1) nowait
for(int i=0; i<n; i++)
    a[i] = bigFunc1(i);

#pragma omp for schedule(dynamic,1)
for(int j=0; j<m; j++)
    b[j] = bigFunc2(j);
```

Credit: IOMPP
Barrier Construct

• Explicit barrier synchronization

• Each thread waits until all threads arrive

```c
#pragma omp parallel shared(A, B, C)
{
    DoSomeWork(A,B); // Processed A into B
    #pragma omp barrier
    DoSomeWork(B,C); // Processed B into C
}
```

Credit: IOMPP
Atomic Construct

- Applies only to simple update of memory location

- Special case of a critical section, to be discussed shortly
  - Atomic introduces less overhead than critical
Atomic Construct

- Applies only to simple update of memory location

- Special case of a \textit{critical} section, to be discussed shortly
  - Atomic introduces less overhead than \textit{critical}

```c
index[0] = 2;
index[1] = 3;
index[2] = 4;
index[3] = 0;
index[4] = 5;
index[5] = 5;
index[6] = 5;
index[7] = 1;
```
Atomic Construct

- Applies only to simple update of memory location

- Special case of a **critical** section, to be discussed shortly
  - Atomic introduces less overhead than **critical**

```c
#pragma omp parallel for shared(x, y, index, n)
for (i = 0; i < n; i++) {
    #pragma omp atomic
    x[index[i]] += work1(i);
    y[i] += work2(i);
}
```

index[0] = 2;
index[1] = 3;
index[2] = 4;
index[3] = 0;
index[4] = 5;
index[5] = 5;
index[6] = 5;
index[7] = 1;
Example: Dot Product

```c
float dot_prod(float* a, float* b, int N) {
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++) {
        sum += a[i] * b[i];
    }
    return sum;
}
```

What is Wrong?
A *race condition* is nondeterministic behavior produced when two or more threads access a shared variable at the same time.

For example, suppose both Thread A and Thread B are executing the statement:

```c
area += 4.0 / (1.0 + x*x);
```
Two Possible Scenarios

Value of area

Thread A  Thread B

11.667
+3.765
15.432
15.432
15.230

18.995

Credit: IOMPP
Two Possible Scenarios

Order of thread execution causes non deterministic behavior in a data race

Credit: IOMPP
Protect Shared Data

- The **critical** construct: protects access to shared, modifiable data
- The critical section allows only one thread to enter it at a given time

```c
float dot_prod(float* a, float* b, int N)
{
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++) {
        #pragma omp critical
        sum += a[i] * b[i];
    }
    return sum;
}
```

Credit: IOMPP
OpenMP Critical Construct

#pragma omp critical [(lock_name)]

- Defines a critical region on a structured block

```c
float RES;
#pragma omp parallel
{
#pragma omp for
for(int i=0; i<niters; i++){
    float B = big_job(i);
    #pragma omp critical (RES_lock)
    consum(B, RES);
}
```

Threads wait their turn – only one at a time calls consum() thereby protecting RES from race conditions.

Naming the critical construct RES_lock is optional but highly recommended.

Includes material from IOMPP
OpenMP Critical Construct

```c
float RES;
#pragma omp parallel
{
#pragma omp for
  for(int i=0; i<niters; i++){
    float B = big_job(i);
    #pragma omp critical (RES_lock)
    consum(B, RES);
  }
}
```

- Defines a critical region on a structured block

Threads wait their turn – only one at a time calls consum() thereby protecting RES from race conditions

Naming the critical construct RES_lock is optional but highly recommended

Good Practice – Name all critical sections
OpenMP Reduction Clause

```
reduction (op : list)
```

- The variables in “list” must be shared in the enclosing parallel region

- Inside parallel or work-sharing construct:
  - A PRIVATE copy of each list variable is created and initialized depending on the “op”
  - These copies are updated locally by threads
  - At end of construct, local copies are combined through “op” into a single value and combined with the value in the original SHARED variable

Credit: IOMPP
Reduction Example

- Local copy of sum for each thread
- All local copies of sum added together and stored in “global” variable

```c
#pragma omp parallel for reduction(+:sum)
  for(i=0; i<N; i++) {
    sum += a[i] * b[i];
  }
```

Credit: IOMPP
OpenMP Reduction Example: Numerical Integration

```c
static long num_steps=100000;
double step, pi;

void main() {
    int i;
    double x, sum = 0.0;

    step = 1.0/(double) num_steps;
    for (i=0; i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
```
OpenMP Reduction Example: Numerical Integration

\[ f(x) = \frac{4.0}{1 + x^2} \]

```c
static long num_steps = 100000;
double step, pi;

void main() {
    int i;
    double x, sum = 0.0;

    step = 1.0/(double) num_steps;
    for (i=0; i< num_steps; i++) {
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n", pi);
}
```

Credit: IOMPP
OpenMP Reduction Example: Numerical Integration

\[ \int_{0}^{1} \frac{4.0}{1+x^2} \, dx = \pi \]

```c
static long num_steps = 100000;
double step, pi;

void main() {
    int i;
double x, sum = 0.0;

    step = 1.0/(double) num_steps;
    for (i=0; i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
```
OpenMP Reduction Example: Numerical Integration

\[ \int_{0}^{4} \frac{4.0}{1 + x^2} \, dx = \pi \]

\begin{verbatim}
static long num_steps=100000;
double step, pi;

void main() {
    int i;
    double x, sum = 0.0;

    step = 1.0/(double) num_steps;
    for (i=0; i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
\end{verbatim}
OpenMP Reduction Example: Numerical Integration

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

int main(int argc, char* argv[]) {
    int num_steps = atoi(argv[1]);
    double step = 1./(double(num_steps));
    double sum;

    #pragma omp parallel for reduction(+:sum)
    {
        for(int i=0; i<num_steps; i++) {
            double x = (i + .5)*step;
            sum += 4.0/(1.+ x*x);
        }
    }

    double my_pi = sum*step;
    printf("Value of integral is: %f\n", my_pi);

    return 0;
}
```
OpenMP Reduction Example:
Output

[negrut@euler24 CodeBits]$ g++ testOMP.cpp -o me964.exe
[negrut@euler24 CodeBits]$ ./me964.exe 100000
Value of integral is: 3.141593
A range of associative operands can be used with reduction
Initial values are the ones that make sense mathematically

<table>
<thead>
<tr>
<th>Operand</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>^</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operand</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>~0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Exercise: Variable Scoping Aspects

• Consider parallelizing the following code

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

int main() {
    const int n = 20;
    int a[n];
    for (int i = 0; i < n; i++)
        a[i] = i;

    // this is the part that needs to
    // be parallelized
    caller(a, n);

    for (int i = 0; i < n; i++)
        printf("a[%d]=%d\n", i, a[i]);

    return 0;
}

void callee(int *x, int *y, int z) {
    int ii;
    static int cv = 0;
    cv++;
    for (ii = 1; ii < z; ii++) {
        *x = *x + *y + z;
    }
    printf("Value of counter: %d\n", cv);
}

void caller(int *a, int n) {
    int i, j, m = 3;
    for (i = 0; i < n; i++) {
        int k = m;
        for (j = 1; j <= 5; j++) {
            callee(&a[i], &k, j);
        }
    }
}
```
Program Output

- Looks good
  - The value of the counter increases each time you hit the "callee" subroutine

- If you run the executable 20 times, you get the same results 20 times
### First Attempt to Parallelize

- Example obviously contrived but helps to understand the scope of different variables

```c
void callee(int *x, int *y, int z) {
    int ii;
    static int cv=0;
    cv++;
    for (ii=1; ii<z; ii++) {
        *x = *x + *y + z;
    }
    printf("Value of counter: %d\n", cv);
}

void caller(int *a, int n) {
    int i, j, m=3;
    #pragma omp parallel for
    for (i=0; i<n; i++) {
        int k=m;
        for (j=1; j<=5; j++) {
            callee(&a[i], &k, j);
        }
    }
}
```

<table>
<thead>
<tr>
<th>Var</th>
<th>Scope</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>shared</td>
<td>Declared outside parallel construct</td>
</tr>
<tr>
<td>n</td>
<td>shared</td>
<td>Declared outside parallel construct</td>
</tr>
<tr>
<td>i</td>
<td>private</td>
<td>Parallel loop index</td>
</tr>
<tr>
<td>j</td>
<td>shared</td>
<td>Declared outside parallel construct</td>
</tr>
<tr>
<td>m</td>
<td>shared</td>
<td>Constant decl. outside parallel construct</td>
</tr>
<tr>
<td>k</td>
<td>private</td>
<td>Automatic variable/parallel region</td>
</tr>
<tr>
<td>x</td>
<td>private</td>
<td>Passed by value</td>
</tr>
<tr>
<td>*x</td>
<td>shared</td>
<td>(actually a)</td>
</tr>
<tr>
<td>y</td>
<td>private</td>
<td>Passed by value</td>
</tr>
<tr>
<td>*y</td>
<td>private</td>
<td>(actually k)</td>
</tr>
<tr>
<td>z</td>
<td>private</td>
<td>(actually j)</td>
</tr>
<tr>
<td>ii</td>
<td>private</td>
<td>Local stack variable in called function</td>
</tr>
<tr>
<td>cv</td>
<td>shared</td>
<td>Declared static (like global)</td>
</tr>
</tbody>
</table>
Program Output, First Attempt to Parallelize

- Looks bad...
  - The values in array “a” are all over the map
  - The value of the counter “cv” changes chaotically within “callee”
  - The function “callee” gets hit a random number of times (should be hit 100 times). Example:
    
    $ parallelGood.exe | grep "Value of counter" | wc -l
    $ 70

- If you run executable 20 times, you get different results

- One of the problems is that “j” is shared
Second Attempt to Parallelize

- Declare the inner loop variable “j” as a private variable within the parallel loop

```c
void callee(int *x, int *y, int z) {
    int ii;
    static int cv=0;
    cv++;
    for (ii=1; ii<z; ii++) {
        *x = *x + *y + z;
    }
    printf("Value of counter: %d\n", cv);
}

void caller(int *a, int n) {
    int i, j, m=3;
    #pragma omp parallel for private(j)
    for (i=0; i<n; i++) {
        int k=m;
        for (j=1; j<=5; j++) {
            callee(&a[i], &k, j);
        }
    }
}
```
Program Output, Second Attempt to Parallelize

- Looks better
  - The values in array “a” are correct
  - The value of the counter “cv” changes strangely within the “callee” subroutine
  - The function “callee” gets hit 100 times:
     $ parallelGood.exe | grep "Value of counter" | wc -l
     $ 100

- If you run executable 20 times, you get good results for “a”, but the static variable will continue to behave strangely (it’s shared)

- Conclusion: be careful when you work with static or some other globally shared variable in parallel programming
  - In general, dealing with such variables is bad programming practice
Slightly Better Solution…

- Declare the inner loop index “j” only inside the parallel segment
  - After all, it’s only used there
  - You get rid of the “private” attribute, less constraints on the code, increasing the opportunity for code optimization at compile time

```c
void callee(int *x, int *y, int z) {
    int ii;
    static int cv=0;
    cv++;
    for (ii=1; ii<z; ii++) {
        *x = *x + *y + z;
    }
    printf("Value of counter: %d\n", cv);
}

void caller(int *a, int n) {
    int i, m=3;
    #pragma omp parallel for
    for (i=0; i<n; i++) {
        int k=m;
        for (int j=1; j<=5; j++) {
            callee(&a[i], &k, j);
        }
    }
}
```

Used here, then you should declare here (common sense…)
Program Output, Parallelized Code

- Looks good
  - The values in array “a” are correct
  - The value of the counter “cv” changes strangely within the “callee” subroutine
  - The function “callee” gets hit 100 times:
    $ parallelGood.exe | grep "Value of counter" | wc -l
    $ 100

- If you run executable 20 times, you get good results for “a”, but the static variable will continue to behave strangely (it’s shared)

- What surprised me: the value of the counter was indeed 100
  - In other words, although shared, no trashing of this variable…
    Still don’t understand why this is the case (it’s surprising it works like this…)
OpenMP API
OpenMP: The 30,000 Feet Perspective
The OpenMP API

- Application Programmer Interface (API) is combination of:
  - Directives
    - Example: `#pragma omp task`
  - Runtime library routines
    - Example: `int omp_get_thread_num(void)`
  - Environment variables
    - Example: `setenv OMP_SCHEDULE "guided, 4"`
The OpenMP API Directives

- Directives (or Pragmas) used to
  - Express/Define parallelism (flow control)
    - Example: `#pragma omp parallel for`
  - Specify data sharing among threads (communication)
    - Example: `#pragma omp parallel for private(x,y)`
  - Synchronization (coordination or interaction)
    - Example: `#pragma omp barrier`
OpenMP 3.1:
Summary of Run-Time Library OpenMP Routines

1. `omp_set_num_threads`
2. `omp_get_num_threads`
3. `omp_get_max_threads`
4. `omp_get_thread_num`
5. `omp_get_thread_limit`
6. `omp_get_num_procs`
7. `omp_in_parallel`
8. `omp_set_dynamic`
9. `omp_get_dynamic`
10. `omp_set_nested`
11. `omp_get_nested`
12. `omp_set_schedule`
13. `omp_get_schedule`
14. `omp_set_max_active_levels`
15. `omp_get_max_active_levels`
16. `omp_get_level`
17. `omp_get_ancestor_thread_num`
18. `omp_get_team_size`
19. `omp_get_active_level`
20. `omp_init_lock`
21. `omp_destroy_lock`
22. `omp_set_lock`
23. `omp_unset_lock`
24. `omp_test_lock`
25. `omp_init_nest_lock`
26. `omp_destroy_nest_lock`
27. `omp_set_nest_lock`
28. `omp_unset_nest_lock`
29. `omp_test_nest_lock`
30. `omp_get_wtime`
31. `omp_get_wtick`
#include <stdio.h>
#include <omp.h>

int main() {
    omp_set_num_threads(4);
    printf_s("First call: %d\n", omp_get_num_threads( ));
    #pragma omp parallel
        #pragma omp master
        {
            printf_s("Second call: %d\n", omp_get_num_threads( ));
        }

    printf_s("Third call: %d\n", omp_get_num_threads( ));

    #pragma omp parallel num_threads(3)
        #pragma omp master
        {
            printf_s("Fourth call: %d\n", omp_get_num_threads( ));
        }

    printf_s("Last call: %d\n", omp_get_num_threads( ));
    return 0;
}
OpenMP: Environment Variables

- **OMP_SCHEDULE**
  - Example: `setenv OMP_SCHEDULE "guided, 4"`

- **OMP_NUM_THREADS**
  - Sets the maximum number of threads to use during execution.
  - Example: `setenv OMP_NUM_THREADS 8`

- **OMP_DYNAMIC**
  - Enables or disables dynamic adjustment of the number of threads available for execution of parallel regions. Valid values are TRUE or FALSE
  - Example: `setenv OMP_DYNAMIC TRUE`

- **OMP_NESTED**
  - Enables or disables nested parallelism. Valid values are TRUE or FALSE
  - Example: `setenv OMP_NESTED TRUE`
OpenMP: Environment Variables
[recent ones, added as of 3.0]

- **OMP_STACKSIZE**
  - Controls the size [in KB] of the stack for created (non-Master) threads.

- **OMP_WAIT_POLICY**
  - Provides hint to an OpenMP implementation about desired behavior of waiting threads.

- **OMP_MAX_ACTIVE_LEVELS**
  - Controls the maximum number of nested active parallel regions. The value of this environment variable must be a non-negative integer. Example:
    - `setenv OMP_MAX_ACTIVE_LEVELS 2`

- **OMP_THREAD_LIMIT**
  - Sets the number of OpenMP threads to use for the whole OpenMP program Example:
    - `setenv OMP_THREAD_LIMIT 8`
OpenMP
Concluding Remarks & Wrap-up
OpenMP Summary

- Shared memory, thread-based parallelism
- Explicit parallelism (relies on you specifying parallel regions)
- Fork/join model

- Industry-standard shared memory programming model
  - First version released in 1997

- OpenMP Architecture Review Board (ARB) determines updates to standard
  - The final specification of Version 3.1 released in July of 2011 (minor update)
OpenMP, Summary

- OpenMP provides small yet versatile programming model
  - This model serves as the inspiration for the OpenACC effort to standardizing approaches that can factor in the presence of a GPU accelerator

- Not at all intrusive, very straightforward to parallelize existing code
  - Good efficiency gains achieved by using parallel regions in an existing code

- Work-sharing constructs: for, section, task enable parallelization of computationally intensive portions of program
Attractive Features of OpenMP

- Parallelize small parts of application, one at a time (beginning with most time-critical parts)

- Can implement complex algorithms

- Code size grows only modestly

- Expression of parallelism flows clearly, code is easy to read

- Single source code for OpenMP and non-OpenMP
  - Non-OpenMP compilers simply ignore OMP directives
OpenMP, Some Caveats

- It seems that the vendors lag behind when it comes to support of the latest OpenMP specifications
  - Intel probably is most up to speed although I haven’t used their compilers

- OpenMP threads are heavy
  - Very good for handling parallel tasks
  - Not particularly remarkable at handling fine grain data parallelism (vector architectures excel here)