DPHPC: Introduction to OpenMP
Recitation session

Based on http://openmp.org/mp-documents/Intro_To_OpenMP_Mattson.pdf
OpenMP – An Introduction

- What is it? A set of compiler directives and a runtime library
  - `#pragma omp parallel num_threads(4)`
  - `#include <omp.h>`

- Why do we care? Simplify (& standardizes) how multi-threaded applications are written
  - Fortran, C, C++
OpenMP – An Introduction

- **OpenMP is based on Fork/Join model**
  - When program starts, one Master thread is created
  - Master thread executes sequential portions of the program
  - At the beginning of parallel region, master thread forks new threads
  - All the threads together now forms a “team”
  - At the end of the parallel region, the forked threads die
What’s a Shared-Memory Program?

- One process that spawns multiple threads
- Threads can communicate via shared memory
  - Read/Write to shared variables
  - Synchronization can be required!
- OS decides how to schedule threads
OpenMP: Hello World

- Make “Hello World” multi-threaded

```c
#include "omp.h"

int main(){
    int ID=0;
    printf("hello(%d) ", ID);
    printf("world(%d)\n", ID);
}
```

```c
#include "omp.h"

int main(){
    #pragma omp parallel
    {
        int ID = omp_get_thread_num();
        printf("hello(%d) ", ID);
        printf("world(%d)\n", ID);
    }
}
```
Parallel Regions

- A parallel region identifies a portion of code that can be executed by different threads
- You can create a parallel region with the “parallel” directive
- You can request a specific number of threads with `omp_set_num_threads(N)`

```c
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    pooh(ID, A);
}
```

- Each thread will call pooh with a different value of ID
Parallel Regions

- All the threads executed the same code
- The A array is shared
- Implicit synchronization at the end of the parallel region
Behind the scenes

- The OpenMP compiler generates code logically analogous to that on the right
- All known OpenMP implementations use a thread pool so full cost of threads creation and destruction is not incurred for each parallel region.
- Only three threads are created because the last parallel section will be invoked from the parent thread.

```c
#pragma omp parallel num_threads(4)
{
    foobar();
}

void thunk()
{
    foobar();
}

pthread_t tid[4];

for (int i = 1; i < 4; ++i)
    pthread_create(&tid[i], 0, thunk, 0);

thunk();

for (int i = 1; i < 4; ++i)
    pthread_join(tid[i]);
```
Exercise: Compute PI

- Mathematically, we know that computing the integral of \( \frac{4}{1+x^2} \) from 0 to 1 will give us the value of \( \pi \) – which is great since it gives us an easy way to check the answer.

- We can approximate the integral as a sum of rectangles:

\[
\sum_{i=0}^{N} F(x_i) \Delta x \approx \pi
\]

- Where each rectangle has width '\( x \) and height \( F(x_i) \) at the middle of interval i.
Computing PI: Sequential Version

```c
#include <stdio.h>
#include <omp.h>
static long num_steps = 10000000;
double step;
int main() {
    int i;
    double x, pi, sum = 0.0;
    double start_time, run_time;
    
    step = 1.0/(double) num_steps;
    
    start_time = omp_get_wtime();

    for (i=1;i<= num_steps; i++){
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }

    pi = step * sum;
    run_time = omp_get_wtime() - start_time;
    printf("\n pi with %ld steps is %lf in %lf seconds\n ",num_steps,pi,run_time);
}
```
Parallel PI

- Create a parallel version of the pi program using a parallel construct.
- Pay close attention to shared versus private variables.

- In addition to a parallel construct:
  - int omp_get_num_threads();
  - int omp_get_thread_num();
  - double omp_get_wtime();

- Possible strategy:
  - Run the same program on P processing elements where P can be arbitrarily large.
  - Use the rank ... an ID ranging from 0 to (P-1) ... to select between a set of tasks and to manage any shared data structures.

This pattern is very general and has been used to support most (if not all) the algorithm strategy patterns.

MPI programs almost always use this pattern ... it is probably the most commonly used pattern in the history of parallel programming.
Simple Parallel PI

```
#include <omp.h>
static long num_steps = 1000000; double step;
#define NUM_THREADS 2
void main ()
{
    int i, nthreads; double pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        int i, id, nthreads;
        double x;
        id = omp_get_thread_num();
        nthreads = omp_get_num_threads();
        if (id == 0) nthreads = nthreads;
        for (i=id, sum[id]=0.0; i<num_steps; i=i+nthreads) {
            x = (i+0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
    }
    for(i=0, pi=0.0; i<nthreads; i++) pi += sum[i] * step;
}
```

Original Serial pi program with 100000000 steps ran in 1.83 seconds.

| threads | 1st
| SPMD |
|--------|------|
| 1      | 1.86 |
| 2      | 1.03 |
| 3      | 1.08 |
| 4      | 0.97 |

Poor scaling!!!
False Sharing

If independent data elements happen to sit on the same cache line, each update will cause the cache lines to “slosh back and forth” between threads.

HotFix: Pad arrays so elements you use are on distinct cache lines.
Padding the PI

```c
#include <omp.h>
static long num_steps = 100000; double step;
define PAD 8 // assume 64 byte L1 cache line size
define NUM_THREADS 2
void main ()
{
    int i, nthreads; double pi, sum[NUM_THREADS][PAD];
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        int i, id, nthrds;
        double x;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        if (id == 0) nthreads = nthrds;
        for (i=id, sum[id]=0.0; i<num_steps; i=i+nthrds) {
            x = (i+0.5)*step;
            sum[id][0] += 4.0/(1.0+x*x);
        }
    }
    for (i=0, pi=0.0; i<nthreads; i++) pi += sum[i][0] * step;
}
```

<table>
<thead>
<tr>
<th>threads</th>
<th>1st SPMD</th>
<th>1st SPMD padded</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.86</td>
<td>1.86</td>
</tr>
<tr>
<td>2</td>
<td>1.03</td>
<td>1.01</td>
</tr>
<tr>
<td>3</td>
<td>1.08</td>
<td>0.69</td>
</tr>
<tr>
<td>4</td>
<td>0.97</td>
<td>0.53</td>
</tr>
</tbody>
</table>

Pad the array so each sum value is in a different cache line.
SPMD vs WorkSharing

- A parallel construct by itself creates an SPMD or “Single Program Multiple Data” program ... i.e., each thread redundantly executes the same code.

- How do you split up pathways through the code between threads within a team?
  - WorkSharing: The OpenMP loop construct (not the only way to go)

- The loop worksharing construct splits up loop iterations among the threads in a team

```c
#pragma omp parallel
{
    #pragma omp for
    for (I=0;I<N;I++)
    {
        NEAT_STUFF(I);
    }
}
```

The variable I is made “private” to each thread by default. You could do this explicitly with a “private(I)” clause.
Why should we use it (the loop construct)?

Sequential:

```c
for(i=0;i<N;i++) { a[i] = a[i] + b[i];}
```

OpenMP parallel region:

```c
#pragma omp parallel
{
    int id, i, Nthrds, istart, iend;
    id = omp_get_thread_num();
    Nthrds = omp_get_num_threads();
    istart = id * N / Nthrds;
    iend = (id+1) * N / Nthrds;
    if (id == Nthrds-1) iend = N;
    for(i=istart;i<iend;i++) { a[i] = a[i] + b[i];}
}
```

OpenMP parallel region and worksharing:

```c
#pragma omp parallel
#pragma omp for
for(i=0;i<N;i++) { a[i] = a[i] + b[i];}
```
Working with Loops

- Basic approach
  - Find compute intensive loops
  - Make the loop iterations independent .. So they can safely execute in any order without loop-carried dependencies
  - Place the appropriate OpenMP directive and test

```c
int i, j, A[MAX];
j = 5;
for (i=0;i< MAX; i++) {
    j +=2;
    A[i] = big(j);
}
```

Note: loop index “i” is private by default

```c
int i, A[MAX];
#pragma omp parallel for
for (i=0;i< MAX; i++) {
    int j = 5 + 2*(i+1);
    A[i] = big(j);
}
```

Remove loop carried dependence
Reduction

- **OpenMP reduction clause:**
  - `reduction (op : list)`
- **Inside a parallel or a work-sharing construct:**
  - A local copy of each list variable is made and initialized depending on the “op” (e.g. 0 for “+”).
  - Updates occur on the local copy.
  - Local copies are reduced into a single value and combined with the original global value.
- **The variables in “list” must be shared in the enclosing parallel region.**

```c
double ave=0.0, A[MAX]; int i;
#pragma omp parallel for reduction (+:ave)
for (i=0;i< MAX; i++) {
    ave += A[i];
}
ave = ave/ MAX;
```
#include <omp.h>
static long num_steps = 100000; double step;
void main()
{
    int i;
    double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;

    #pragma omp parallel
    {
        double x;
        #pragma omp for reduction(+:sum)
        for (i=0; i<num_steps; i++)
        {
            x = (i+0.5)*step;
            sum = sum + 4.0/(1.0+x*x);
        }
    }

    pi = step * sum;
}