CSCI-UA.0480-003
Parallel Computing

Lecture 12: OpenMP - I

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com
Small and Easy Motivation

#include <stdio.h>
#include <stdlib.h>

int main() {

    // Do this part in parallel

    printf( "Hello, World!\n" );

    return 0;
}
Small and Easy Motivation

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

int main() {

  omp_set_num_threads(16);

  // Do this part in parallel
  #pragma omp parallel
  {
    printf( "Hello, World!\n" );
  }

  return 0;
}
```
Simple!

OpenMP can parallelize many serial programs with relatively few annotations that specify parallelism and independence.

OpenMP is a small API that hides cumbersome threading calls with simpler directives.
Interesting Insights About OpenMP

These insights are coming from HPC folks though!

Source: www.sdsc.edu/~allans/cs260/lectures/OpenMP.ppt
OpenMP

- An **API** for shared-memory parallel programming.
- Designed for systems in which each thread can potentially have access to all available memory.
- System is viewed as a collection of cores or CPU’s, all of which have access to the same main memory → shared memory architecture
Pragmas

- Special **preprocessor** instructions.
- specified by the C standard for providing additional information to the compiler, beyond what is conveyed in the language itself.
- Compilers that don’t support the pragmas ignore them.

#pragma
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void Hello(void); /* Thread function */

int main(int argc, char* argv[]) {
  /* Get number of threads from command line */
  int thread_count = strtol(argv[1], NULL, 10);

  #pragma omp parallel num_threads(thread_count)
  Hello();

  return 0;
} /* main */

void Hello(void) {
  int my_rank = omp_get_thread_num();
  int thread_count = omp_get_num_threads();

  printf("Hello from thread %d of %d\n", my_rank, thread_count);
}
/* Hello */
gcc -g -Wall -fopenmp -o omp_hello omp_hello.c

./omp_hello 4

running with 4 threads

Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4

Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4

Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4

possible outcomes
Another Hello World

- One of the possible execution sequences:

```c
int main (int argc, char **argv)
{
    int numThr = atoi (argv[1]);

    #pragma omp parallel num_threads(numThr)
    cout << "Hello from thread " <<
        Omp_get_thread_num () << endl;

    return 0;
}
```

Child thread spawning

End-of-execution signal

Child thread termination

Parallel region
OpenMp pragmas

- `# pragma omp parallel`
  
  - Most basic parallel directive.
  - The number of threads that run the following *structured block of code* is determined by the run-time system.
A process forking and joining two threads
clause

- Text that modifies a directive.
- The `num_threads` clause can be added to a parallel directive.
- It allows the programmer to specify the number of threads that should execute the following block.

```c
#pragma omp parallel num_threads ( thread_count )
```
Of note...

- There may be system-defined limitations on the number of threads that a program can start.
- The OpenMP standard doesn’t guarantee that this will actually start `thread_count` threads.
- Unless we’re trying to start a lot of threads, we will almost always get the desired number of threads.
Some terminology

• In OpenMP parlance the collection of threads executing the parallel block — the original thread and the new threads — is called a **team**, the original thread is called the **master**, and the additional threads are called **slaves**.
Again: The trapezoidal rule
Serial algorithm

/* Input: a, b, n */
h = (b−a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n−1; i++) {
    x_i = a + i*h;
approx += f(x_i);
}
approx = h*approx;
1) We identified two types of tasks:
   a) computation of the areas of individual trapezoids, and
   b) adding the areas of trapezoids.

2) There is no communication among the tasks in the first collection, but each task in the first collection communicates with task 1b.
3) We assumed that there would be many more trapezoids than cores.

- So we aggregated tasks by assigning a contiguous block of trapezoids to each thread.
Assignment of trapezoids to threads
Unpredictable results when two (or more) threads attempt to simultaneously execute:

```python
global_result += my_result;
```
Mutual exclusion

```c
#include <omp.h>

#pragma omp critical
global_result += my_result;
```

only one thread can execute
the following structured block at
a time
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void Trap(double a, double b, int n, double* global_result_p);

int main(int argc, char* argv[]) {
    double global_result = 0.0;  /* Store result in global_result */
    double a, b;                /* Left and right endpoints */
    int n;                      /* Total number of trapezoids */
    int thread_count;

    thread_count = strtol(argv[1], NULL, 10);
    printf("Enter a, b, and n\n\n");
    scanf("%lf %lf %d", &a, &b, &n);
    #pragma omp parallel num_threads(thread_count)
    Trap(a, b, n, &global_result);

    printf("With n = %d trapezoids, our estimate\n", n);
    printf("of the integral from %f to %f = %.14e\n", 
        a, b, global_result);
    return 0;
} /* main */
void Trap(double a, double b, int n, double* global_result_p) {
    double h, x, my_result;
    double local_a, local_b;
    int i, local_n;
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    h = (b-a)/n;
    local_n = n/thread_count;
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    my_result = (f(local_a) + f(local_b))/2.0;
    for (i = 1; i <= local_n-1; i++) {
        x = local_a + i*h;
        my_result += f(x);
    }
    my_result = my_result*h;

    # pragma omp critical
        *global_result_p += my_result;
}/*! Trap. */
Another Example

• **Problem:** Count the number of times each ASCII character occurs on a page of text.

• **Input:** ASCII text stored as an array of characters.

• **Output:** A histogram with 128 buckets – one for each ASCII character

[source: http://www.futurechips.org/tips-for-power-coders/writing-optimizing-parallel-programs-complete.html]
void compute_histogram_st(char *page, int page_size, int *histogram) {
    for (int i = 0; i < page_size; i++) {
        char read_character = page[i];
        histogram[read_character]++;
    }
}
Another Example

We need to parallelize this.

source: http://www.futurechips.org/tips-for-power-coders/writing-optimizing-parallel-programs-complete.html
Another Example

```c
1: void compute_histogram_st(char *page, int page_size, int *histogram) {
2:     #pragma omp parallel for
3:     for (int i = 0; i < page_size; i++) {
4:         char read_character = page[i];
5:         histogram[read_character]++;
6:     }
```

The above code does not work!! Why?

source: http://www.futurechips.org/tips-for-power-coders/writing-optimizing-parallel-programs-complete.html
Another Example

1: void compute_histogram_mt2(char *page, int page_size, int *histogram){
2: #pragma omp parallel for
3: for(int i = 0; i < page_size; i++){
4:      char read_character = page[i];
5:      #pragma omp atomic
6:      histogram[read_character]++;
7:  }
8: }

Speed on Quad Core:
114.89 seconds
> 10x slower than the single thread version!!

source: http://www.futurechips.org/tips-for-power-coders/writing-optimizing-parallel-programs-complete.html
Another Example

```c
void compute_histogram_mt(char *page,
                          int page_size,
                          int *histogram, int num_buckets){

#pragma omp parallel
{
  int local_histogram[111][num_buckets];
  int tid = omp_get_thread_num();

#pragma omp for nowait
  for(int i = 0; i < page_size; i++){
    char read_character = page[i];
    local_histogram[tid][read_character]++;
  }

#pragma omp atomic
  for(int i = 0; i < num_buckets; i++){
    histogram[i] += local_histogram[tid][i];
  }
}
```

source: http://www.futurechips.org/tips-for-power-coders/writing-optimizing-parallel-programs-complete.html
void compute_histogram_mt4(char *page, int page_size,  
     int *histogram, int num_buckets){
1:       int num_threads = omp_get_max_threads();
2:       #pragma omp parallel
3:       {
4:          __declspec (align(64)) int local_histogram[num_threads+1][num_buckets];
5:          int tid = omp_get_thread_num();
6:          #pragma omp for
7:          for(int i = 0; i < page_size; i++){
8:             char read_character = page[i];
9:             local_histogram[tid][read_character]++;
10:          }
11:          #pragma omp barrier
12:          #pragma omp single
13:          for(int t = 0; t < num_threads; t++){
14:             for(int i = 0; i < num_buckets; i++)
15:                histogram[i] += local_histogram[t][i];
16:          }
17:      }

Speed is 4.42 seconds.  
Slower than the previous version.
Another Example

```c
void compute_histogram_mt4(char *page, int page_size,
                          int *histogram, int num_buckets){

  1:    int num_threads = omp_get_max_threads();
  2:    #pragma omp parallel
  3:    {
  4:      __declspec (align(64)) int local_histogram[num_threads+1][num_buckets];
  5:      int tid = omp_get_thread_num();
  6:      #pragma omp for
  7:      for(int i = 0; i < page_size; i++){
  8:          char read_character = page[i];
  9:          local_histogram[tid][read_character]++;
 10:      }

 11:    #pragma omp for
 12:    for(int i = 0; i < num_buckets; i++){
 13:        for(int t = 0; t < num_threads; t++)
 14:            histogram[i] += local_histogram[t][i];
 15:    }

 16: }

Speed is 3.60 seconds.
```

**source:** http://www.futurechips.org/tips-for-power-coders/writing-optimizing-parallel-programs-complete.html
What Can We Learn from the Previous Example?

• Atomic operations
  – They are expensive
  – Yet, they are fundamental building blocks.

• Synchronization:
  – correctness vs performance loss
  – Rich interaction of hardware-software tradeoffs
  – Must evaluate hardware primitives and software algorithms together
1. Start with a parallelizable algorithm
   • loop-level parallelism is necessary
2. Implement serially
3. Test and Debug
4. Annotate the code with parallelization (and synchronization) directives
   • Hope for linear speedup
5. Test and Debug
All OpenMP programs begin with a single thread: **master thread** (ID = 0). 

**FORK:** the master thread then creates a team of parallel **threads**.

**JOIN:** When the team threads complete the statements in the parallel region construct, they synchronize and terminate.
int main() {
  // serial region
  printf("Hello...");
  // parallel region
  #pragma omp parallel
  {
    printf("World");
  }
  // serial again
  printf("!");
}

We didn’t use `omp_set_num_threads()`, what will be the output?
Isn’t Nested Parallelism Interesting?
Important!

• The following are implementation dependent:
  – Nested parallelism
  – Dynamically alter number of threads

• It is entirely up to the programmer to ensure that I/O is conducted correctly within the context of a multithreaded program.
What we learned so far

- `#include <omp.h>`
- `gcc -fopenmp ...`
- `omp_set_num_threads(x);`
- `omp_get_thread_num();`
- `omp_get_num_threads();`
- `#pragma omp parallel [num_threads(x)]`
- `#pragma omp parallel for [nowait]`
- `#pragma omp atomic`
- `#pragma omp barrier`
- `#pragma omp single`
- `#pragma omp critical`
Conclusions

- OpenMP is a standard for programming shared-memory systems.
- OpenMP uses both special functions and preprocessor directives called pragmas.
- OpenMP programs start multiple threads rather than multiple processes.
- Many OpenMP directives can be modified by clauses.