

# COSC 407

## Intro to Parallel Computing

Topic 5 - Introduction to OpenMP

COSC 407: Intro to Parallel Computing

1

## Outline

### *Previously:*

- Intro to parallel computing
  - important concepts and terminology.
- Intro to POSIX Threads

### *Today:*

- OpenMP
  - Basics, HelloWorld
  - Distributing the work
  - Example: Summing it all up (as array)

Topic 5: Intro to OpenMP

COSC 407: Intro to Parallel Computing

2

# Roadmap

- Writing programs that use OpenMP.
- Exploiting a powerful OpenMP feature
  - Parallelize serial for loops with only small changes to the source code.
- Using other OpenMP features:
  - Task parallelism.
  - Explicit thread synchronization.
- Addresses standard problems in shared-memory programming

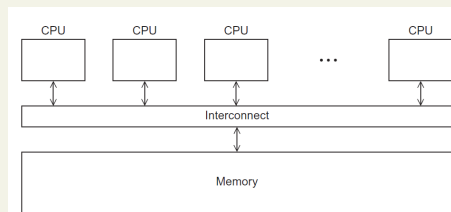
Topic 5: Intro to OpenMP

COSC 407: Intro to Parallel Computing

3

# Shared Memory

- OpenMP = Open Multi-Processing
- An Application Program Interface (API) for multi-threaded, shared-memory parallel programming.
  - Designed for systems in which each thread or process 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 main memory.



Topic 5: Intro to OpenMP

COSC 407: Intro to Parallel Computing

4



## OpenMP vs Pthread

- OpenMP
  - Higher level
  - Programmer only states that a block of code is to be executed in parallel and delegates “how to” to the compiler & runtime
  - Requires compiler support (some compilers cannot compile OpenMP programs)
- Pthreads
  - Lower level
  - Requires programmer to explicitly specify behavior of each thread
  - Library of functions to be linked to a C program (can use any compiler)

Topic 5: Intro to OpenMP

COSC 407: Intro to Parallel Computing

5



## Advantages of OpenMP

- Good performance and scalability
  - If you do it right
- Requires little programming effort (relatively!)
- De-facto and mature standard
- Portable program (large number of compilers)
- Allows the program to be parallelized incrementally
- Ideally suited for multicore

Topic 5: Intro to OpenMP

COSC 407: Intro to Parallel Computing

6



## Compiling and Running From the Command Line

```
gcc -g -Wall -fopenmp -o omp_hello omp_hello.c
```

```
./omp_hello
```

compiling

running (# of threads set with OpenMP env var)

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

possible  
outputs

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

Topic 5: Intro to OpenMP

COSC 407: Intro to Parallel Computing

7

## Compiling and Running with Eclipse

### On Eclipse for Scientific Computing:

- Make sure your MinGW is installed and properly configured.
- Create your code:
  - Create a new “C Project” > “OpenMP Empty C Project”
  - Then create a new “Source File”
- Build your project
  - I created a shortcut for that!
- Run your project
  - Default shortcut: Ctrl+F11
  - Eclipse will take care of the flags with the command line.

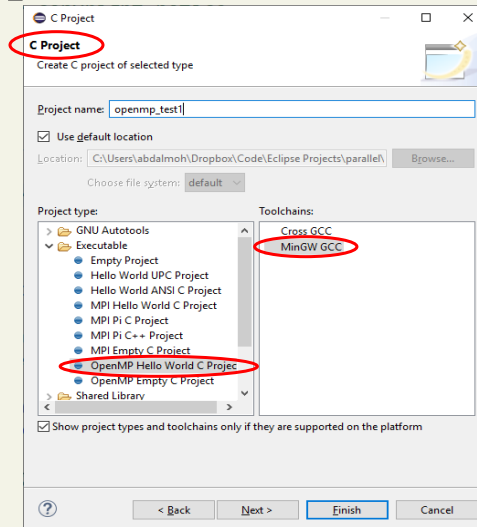
- **Note that you must have the C compiler, openMP, and Eclipse installed properly (See Canvas)**

Topic 5: Intro to OpenMP

COSC 407: Intro to Parallel Computing

8

# Compiling and Running with Eclipse



Topic 5: Intro to OpenMP

COSC 407: Intro to Parallel Computing

9



## Remember!

Two ways to divide the work:

### 1. Task parallelism

- Partition various tasks carried out solving the problem among the cores.



Questions 1 - 5  
All exams



Questions 6 - 10  
All exams



Questions 11 - 15  
All exams

### 1. Data parallelism

- Partition the data used in solving the problem among the cores.
- Each core carries out **similar operations** on it's part of the data.



All tasks (questions)  
1/3 of exams



All tasks (questions)  
1/3 of exams



All tasks (questions)  
1/3 of exams

Topic 5: Intro to OpenMP

COSC 407: Intro to Parallel Computing

10



## OpenMP API

- OpenMP specifications:
  - <http://www.openmp.org/specifications>
- OpenMP is based on **directives**
  - Simple syntax
  - **Risk:** you must understand exactly what the compiler will do, otherwise your program will not function as expected

OpenMP API has **three components**:

1. **Compiler directives (#)**
  - e.g., `#pragma omp parallel`
2. **Runtime library routines**
  - e.g., `omp_get_thread_num()`
3. **Environment variables**
  - E.g., setting an environment variable  
`set OMP_NUM_THREADS=3`
  - Will rarely use or refer to them (as we can set number of threads directly in code)

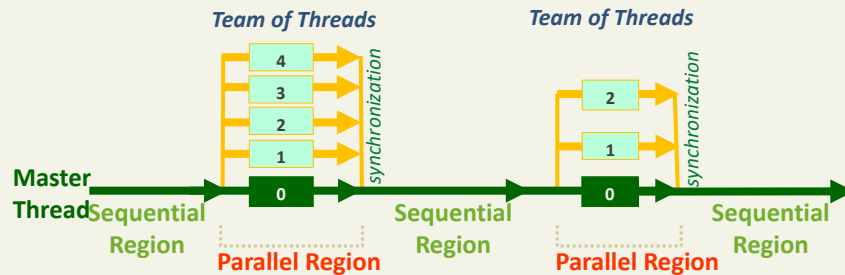


## OpenMP and Threads

- OpenMP implements parallelism exclusively using **threads**
- Remember Process vs Thread:
  - Threads exist within a process.
  - **Execution:**
    - Both threads and processes are **units of execution** (tasks)
  - **Memory:**
    - **Processes** will, by default, **not share memory**.
    - **Threads** of the same process will, by default, have access to the same **shared memory** (the process memory)
      - Light weight

## ★ Fork - Join Model

- OpenMP uses **Fork-Join** model.



- Synchronization** means that everyone must wait till everyone is finished before proceeding to the next region.
- The collection of threads executing the parallel block is called a **team**: the original thread is called the **master**, and the additional threads are called **slaves**
- Each thread takes an **ID** (master always has ID 0)

Topic 5: Intro to OpenMP

COSC 407: Intro to Parallel Computing

13

## ★ OpenMP: pragmas

- Special preprocessor instructions that are added to a system to allow behaviors that aren't part of the basic C specification.
  - Compilers that don't support the pragmas ignore them.
    - i.e. if the compiler does not support them, the program will still yield correct behavior, but without any parallelism.

- Syntax:

**#pragma omp directive [clause[clause]..]**

- Directive**: specifies the required directive
  - The most basic directive: **#pragma omp parallel**
    - The parallel construct forms a team of threads and starts parallel execution.
- Clause**: information to modify the directive.
  - e.g.: **#pragma omp parallel num\_threads(10)**
- Continuation: use \ in pragma
  - e.g. **#pragma omp parallel \**  
**num\_threads(10)**

Topic 5: Intro to OpenMP

COSC 407: Intro to Parallel Computing

14



## 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.
- There is an **implicit barrier** at the end of this directive
  - barriers are discussed later ....

### # pragma omp parallel num\_threads( thread\_count )

- **num\_threads clause** allows the programmer to specify the number of threads that should execute the following block
- 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
- Most current systems can start hundreds or even thousands of threads Unless we're trying to start a lot of threads, we will almost always get the desired number of threads

Topic 5: Intro to OpenMP

COSC 407: Intro to Parallel Computing

15

## Hello World Serial Version

```
#include <stdio.h>

int main() {
    printf("Hello World!\n");
    return 0;
}
```

**Output:**  
Hello World!

Topic 5: Intro to OpenMP

COSC 407: Intro to Parallel Computing

16



# Hello World Parallel Version

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

int main() {
    #pragma omp parallel
    printf("Hello World!\n");
    return 0;
}
```

**Parallel Region.**  
Sent to all threads!

## Output:

Hello World!  
Hello World!  
Hello World!  
Hello World!

Topic 5: Intro to OpenMP

COSC 407: Intro to Parallel Computing

17

# Both Serial and Parallel in One Program

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

int main() {
    printf("Hello Sequential!\n");
    #pragma omp parallel
    printf("Hello Parallel!\n");
    return 0;
}
```

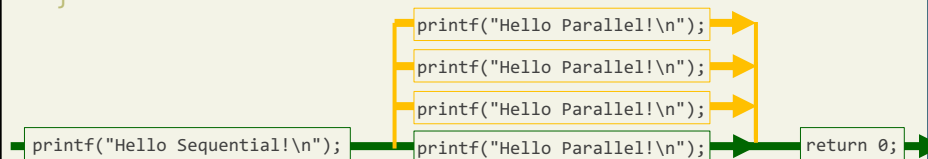
## Output:

Hello Sequential!  
Hello Parallel!  
Hello Parallel!  
Hello Parallel!  
Hello Parallel!

**Sequential :** Only performed by master!

**Parallel:** SAME task sent to all threads

**Sequential**



Topic 5: Intro to OpenMP

COSC 407: Intro to Parallel Computing

18

# Hello World

## Parallel Version (again)

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

### Printing the thread ID

```
int main() {
    #pragma omp parallel
    {
        int my_id = omp_get_thread_num();
        printf("Hello World from thread %d\n", my_id);
    }
    return 0;
}
```

*Use curly braces when  
having more than one  
statement*

#### Possible Output:

```
Hello World from thread 2
Hello World from thread 0
Hello World from thread 1
Hello World from thread 3
```

Topic 5: Intro to OpenMP

COSC 407: Intro to Parallel Computing

19

# Hello World

## Parallel Version (again), cont'd

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

### Printing the thread ID and total num of threads

```
int main() {
    #pragma omp parallel
    {
        int my_id = omp_get_thread_num();
        int tot = omp_get_num_threads();
        printf("Hello World from thread %d/%d\n", my_id, tot);
    }
    return 0;
}
```

#### Possible Output:

```
Hello World from thread 1/4
Hello World from thread 0/4
Hello World from thread 2/4
Hello World from thread 3/4
```

Topic 5: Intro to OpenMP

COSC 407: Intro to Parallel Computing

20

# Hello World

## Parallel Version (again), *cont'd*

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

int main() {
    #pragma omp parallel num_threads(3)
    {
        int my_id = omp_get_thread_num();
        int tot = omp_get_num_threads();
        printf("Hello World from thread %d/%d\n", my_id, tot);
    }
    return 0;
}
```

Specifying  
required # of  
threads

### Possible Output:

```
Hello World from thread 2/3
Hello World from thread 0/3
Hello World from thread 1/3
```

Topic 5: Intro to OpenMP

COSC 407: Intro to Parallel Computing

21

# Hello World

## Parallel Version (again), *cont'd*

```
#include <stdio.h>
#include <omp.h>
void Hello();
int main() {
    #pragma omp parallel num_threads(3)
    {
        Hello();
    }
    return 0;
}

void Hello(){
    int my_id = omp_get_thread_num();
    int count = omp_get_num_threads();
    printf("Hello World from thread %d/%d\n", my_id, count);
}
```

Using a  
function in the  
parallel region

### Possible Output:

```
Hello World from thread 2/3
Hello World from thread 0/3
Hello World from thread 1/3
```

Topic 5: Intro to OpenMP

COSC 407: Intro to Parallel Computing

22

# Demo

## Interleaved Execution

```
#pragma omp parallel num_threads(4)
{
    printf("T%d:A \n", omp_get_thread_num());
    printf("T%d:B \n", omp_get_thread_num());
}
printf("T%d:Done \n", omp_get_thread_num());
```

### Notes:

- Output is *interleaved*, BUT any thread has to print A first then B.
- "Done" is only printed after everyone is done (*synchronization*)

possible outputs

|         |         |         |         |
|---------|---------|---------|---------|
| T0:A    | T1:A    | T1:A    | T2:A    |
| T0:B    | T2:A    | T0:A    | T2:B    |
| T1:A    | T1:B    | T1:B    | T1:A    |
| T1:B    | T2:B    | T0:B    | T1:B    |
| T2:A    | T0:A    | T3:A    | T0:A    |
| T2:B    | T0:B    | T3:B    | T0:B    |
| T3:A    | T3:A    | T2:A    | T3:A    |
| T3:B    | T3:B    | T2:B    | T3:B    |
| T0:Done | T0:Done | T0:Done | T0:Done |

Topic 5: Intro to OpenMP

COSC 407: Intro to Parallel Computing

23

## ★ Distributing Tasks

- You can use Thread ID to *assign tasks* to different threads

```
#pragma omp parallel num_threads(4)
{
    int id = omp_get_thread_num();

    printf("T%d:A\n", id);
    printf("T%d:B\n", id);

    if(id == 2)
        printf("T2:special task\n");
}
printf("End");
```

### Possible output

```
T2:A
T1:A
T2:B
T2:special task
T1:B
T0:A
T0:B
T3:A
T3:B
End
```

### Possible output

```
T0:A
T2:A
T0:B
T2:B
T2:special task
T1:A
T1:B
T3:A
T3:B
End
```

### Notes:

- Only T2 runs the special task
- No specific order for the threads, BUT the statements in the same thread are ordered

Topic 5: Intro to OpenMP

COSC 407: Intro to Parallel Computing

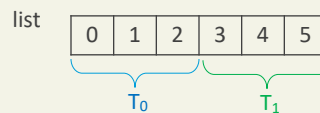
24



## Distributing Data

- You can also use Thread ID to **distribute data** over different threads

```
int list[6] = {0,1,2,3,4,5};
#pragma omp parallel num_threads(2)
{
    int id = omp_get_thread_num();
    int my_a = id * 3;
    int my_b = id * 3 + 3;
    printf("T%d will process indexes %d to %d\n",id,my_a,my_b-1);
    for (int i = my_a; i < my_b; i++)
        printf("T%d:processing list[%d]\n",id,i);
}
printf("End");
return 0;
```



### Possible Output 1:

```
T0 will process indexes 0 to 2
T0:processing list[0]
T0:processing list[1]
T0:processing list[2]
T1 will process indexes 3 to 5
T1:processing list[3]
T1:processing list[4]
T1:processing list[5]
End
```

### Possible Output 2:

```
T0 will process indexes 0 to 2
T1 will process indexes 3 to 5
T0:processing list[0]
T1:processing list[3]
T0:processing list[1]
T1:processing list[4]
T0:processing list[2]
T1:processing list[5]
End
```

Topic 5: Intro to OpenMP

COSC 407: Intro to Parallel Computing

25

## A Toy Problem

- The aim is to compute the sum of values in an array
  - While this is a straight-forward problem, it highlights the key things that need to be considered when dealing with parallelization
- One way to do this is to divide the array of values into a series of sub-arrays
  - Find the sum of each
  - Then sum totals from sub-array

Topic 5: Intro to OpenMP

COSC 407: Intro to Parallel Computing

26

# The Serial Algorithm

```
#include<stdio.h>

int arr[10] = { 12, 1324, 3243, 43, 44, 88, 12333, 34, 33, 4};

int main()
{
    int sum = 0;
    int size = sizeof(arr);

    //add all elements to the variable sum.
    for(int i = 0; i < size; i++)
        sum = sum + arr[i]; // same as sum += arr[i];

    //print the result
    printf("Sum of the array = %d\n",sum);

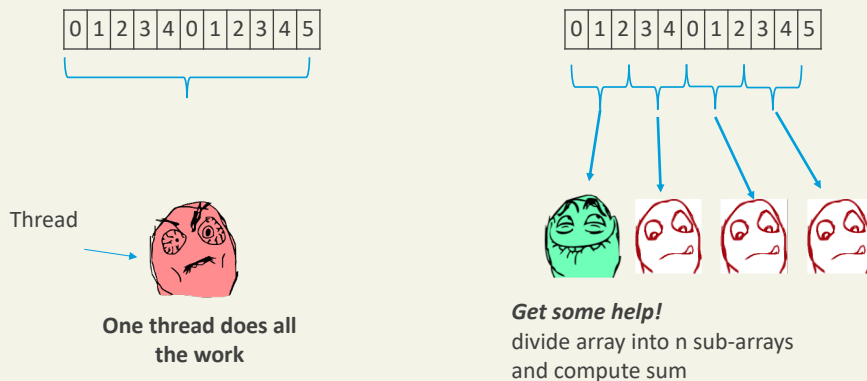
    return 0;
}
```

Topic 5: Intro to OpenMP

COSC 407: Intro to Parallel Computing

27

# Dividing the Work!



Topic 5: Intro to OpenMP

COSC 407: Intro to Parallel Computing

28

## Sum Calculation: Parallel Version

1. Two types of tasks:
  - a) Computation of the sum of sub-arrays
  - b) Adding the partial sums to compute total sum
2. There is no communication among the tasks in 1(a) (assuming that each thread can access the data separately) **but tasks communicates in 1(b) (bringing results back together)**
  - **Potential issues?**
3. We want to assign a single **thread to each core**
  - There could be more sub-arrays than physical cores....

Topic 5: Intro to OpenMP

COSC 407: Intro to Parallel Computing

29

## Sum Calculation: Parallel Version, cont'd

- Manually divide the work among threads
  - Use the “thread ID” and “thread count” to:
    - Compute local values for your calculations
    - Make decisions about which thread executes code
    - In this example,
      - Use “thread count” to calculate number of array slices
      - Use thread ID to determine start and end of each sub-array processed by that thread
  - Ensure no two threads perform the same calculations twice
- Use a private (local) variable to hold the local results and eventually either return them or added them to a global variable

Topic 5: Intro to OpenMP

COSC 407: Intro to Parallel Computing

30

## Serial Code

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

int *arr;
int n = 1000000; //number of enteries in array

int serial_sum(int * arr, int size)
{
    int sum = 0;
    for (int i = 0; i < size; i++)
    {
        sum = sum + arr[i];
    }
    return sum;
}

int main()
{
    int sum = 0;
    int *arr = malloc(n * sizeof(int));
    //generate random data
    for (int i = 0; i < n; i++)
        arr[i] = rand();

    // start the timer!
    double start_time = omp_get_wtime();
    sum = serial_sum(arr, n);
    double end_time = omp_get_wtime();

    printf("Sum of the array = %d in %f ms!\n", sum, (end_time-start_time)*1000);
    free(arr); //housekeeping!

    return 0;
}
```

Time: 1.735000 ms  
n = 1000000

To measure  
execution time

Topic 5: Intro to OpenMP

COSC 407: Intro to Parallel Computing

31

## Parallel Code

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

#define NUM_THREADS 8

int *arr;
int n = 100; //number of enteries in array

void parallel_sum(int* arr, int* global_sum);

int main(int argc, char *argv[])
{
    if (argc > 1){n = atoi(argv[1]);} //no error checking here!!!

    int sum = 0;
    int *arr = malloc(n * sizeof(int));
    //generate random data
    for (int i = 0; i < n; i++)
    {
        arr[i] = rand();
    }

    int global_sum = 0;
    // start the timer!
    double start_time = omp_get_wtime();
#pragma omp parallel num_threads(NUM_THREADS)
    parallel_sum(arr, &global_sum);

    double end_time = omp_get_wtime();

    printf("Parallel sum of the array = %d in %f ms!\n", global_sum, (end_time-start_time)*1000);
    free(arr); //housekeeping!

    return 0;
}
```

Time: 0.564000 ms  
n = 1000000

Topic 5: Intro to OpenMP

COSC 407: Intro to Parallel Computing

32



## Parallel Code cont.

```
void parallel_sum(int* arr, int* global_sum)
{
    int my_id = omp_get_thread_num();    //0, 1, 2, or ...
    int my_n = n / NUM_THREADS;          //# of elements by this thread
    int start = my_id * my_n;            //first element processed
    int end = start + my_n - 1;          //last element processed
    int my_sum = 0;

#ifdef DEBUG
    printf("Thread %d is starting at %d, ending at %d, processing %d\n", my_id, start, end, my_n);
#endif

    for(int i = start; i <= end; i++)
    {
        my_sum += arr[i];
    }

    #pragma omp critical
    *global_sum += my_sum;
}
```

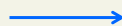
Topic 5: Intro to OpenMP

COSC 407: Intro to Parallel Computing

33

## What if the Compiler Doesn't Support OpenMP?

# include <omp.h>



```
#ifdef _OPENMP
# include <omp.h>
#endif
```

```
# ifdef _OPENMP
    int my_id = omp_get_thread_num ( );
    int thread_count = omp_get_num_threads ( );
# else
    int my_id = 0;
    int thread_count = 1;
# endif
```

Topic 5: Intro to OpenMP

COSC 407: Intro to Parallel Computing

34

## Conclusion/Up Next

- What we covered today (review key concepts):

- Intro to OpenMP
  - Basics, HelloWorld
  - Distributing the work
  - Example: Summing in up

- Next:

- Mutual Exclusion (critical, atomic, locks)
- Variable scope (shared, private, firstprivate)
- Reduction
- Synchronization (barriers, nowait)

## Homework

- Please review

- OpenMP Resources (See week three module)
- Additional resources on Canvas
- Run the sample code and try the challenge
  - You need to be able to run and understand how to approach a problem