

CS4172L: Parallel and Distributed Computing Lab

Lab 1+2: Introduction Lab + Parallel 'Directives'
for Compiler

Open Multi-Processing (OpenMP)

- OpenMP stands for Open Multi-Processing. 'Open' stands for Open source. Multi-Processing means Multi Threading. This means OpenMP is open source API which is used for multi-threading.
- OpenMP is an Application Program Interface (API) designed for parallel programming in shared-memory systems. It allows developers to write parallel code that can be executed concurrently by multiple threads within a single process. The primary goal of OpenMP is to provide a simple and portable way to express parallelism in programs.

Key Points: Open Multi-Processing (OpenMP)

Parallel Programming Model:

- OpenMP is a parallel programming model and API for shared-memory architectures. It provides a set of compiler directives and library functions that enable the creation of parallel programs.

Multi-Threaded Execution:

- OpenMP is particularly well-suited for multi-threaded execution on a single machine. It allows developers to parallelize loops, sections, and tasks easily, enabling efficient utilization of multiple processor cores.

Compiler Directives:

- OpenMP uses special compiler directives (pragmas) to indicate which parts of the code should be parallelized. These directives guide the compiler in generating parallel code.

Open Multi-Processing (OpenMP)

- It is managed by consortium OpenMP Architecture Review Board (OpenMP ARB) which is formed by several companies like AMD, Intel, IBM, HP, Nvidia, Oracle etc.
- OpenMP is available for languages C, C++, Fortran. For the first time, OpenMP ARB releases OpenMP for Fortran language in 1997. In the next year, it gave OpenMP API for C/C++ languages.

Compiler Support and Installation

- Many modern compilers support OpenMP, including GCC (GNU Compiler Collection), Clang, Intel Compiler, Microsoft Visual C++, and others. The compiler must be OpenMP-enabled to recognize and process OpenMP directives.
- **GCC (Windows):** For Windows, you can use MinGW-w64, which provides GCC for Windows. You can download it from MinGW-w64 website.
- **Microsoft Visual C++ (Windows):** Microsoft Visual C++ includes OpenMP support. Make sure you have a recent version of Visual Studio installed.

Thread and Shared Memory Model

- Each process starts with one main thread. This thread is called Master Thread in OpenMP. For a particular block of code, we create multiple threads along with this master thread. These extra threads other than master thread are called as Slave Threads.
- OpenMP is called as Shared Memory model as OpenMP is used to create multiple threads and these multiple threads share the memory of main process.
- OpenMP is called as Fork-Join model. It is because, all slave threads after execution get joined to the master thread. i.e. Process starts with single master thread and ends with single master thread.

Experiments

- **Parallel Loop:**

- a team of parallel threads is created, and each thread executes a portion of the loop iterations.

- **Task Parallelism:**

- Task parallelism focuses on breaking down a task into smaller, independent tasks, and the OpenMP runtime dynamically schedules these tasks among available threads.

- **Parallel Sections:**

- Parallel sections focus on dividing the code into different sections, and each section is executed by a separate thread. The sections are statically assigned to threads at the beginning of the parallel region.

Experiments

- **Data Sharing:**

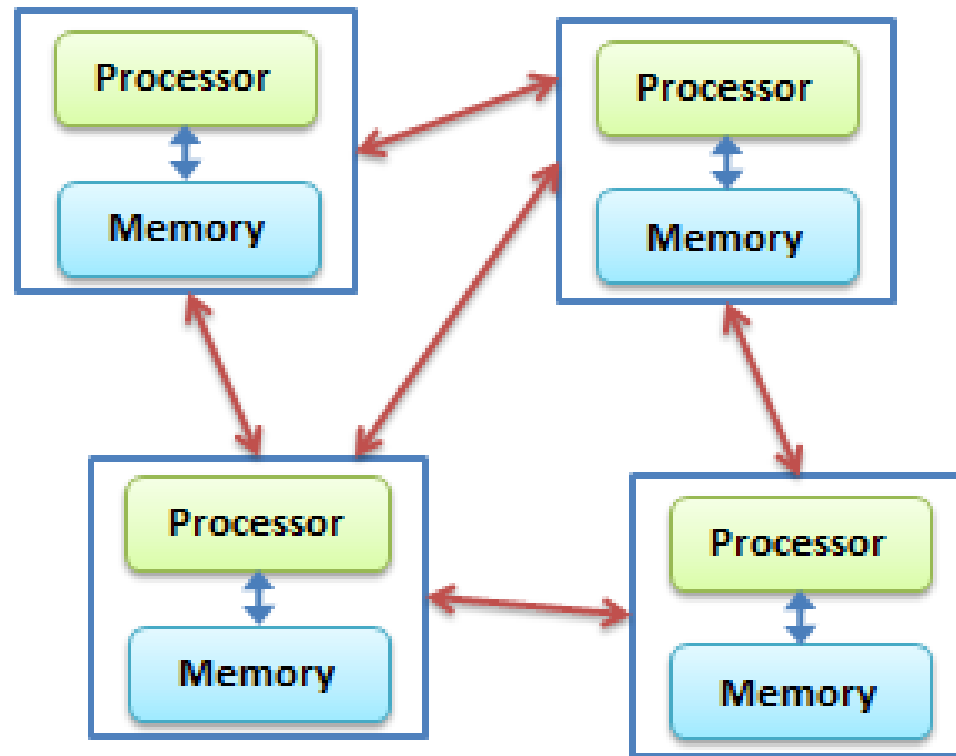
- Data sharing constructs determine how variables are shared or privatized among threads in a parallel region.

- **Work Sharing Constructs:**

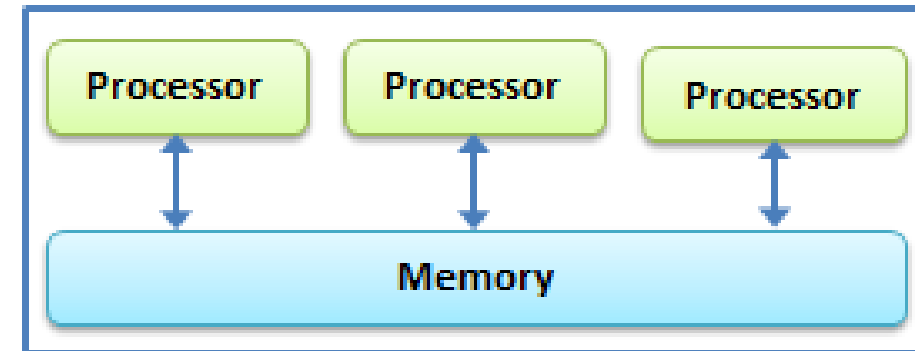
- Work sharing constructs distribute work among threads, allowing them to execute different parts of the code concurrently. (Like parts of a question paper among different evaluators).

Distributed Computing

Distributed Computing



Parallel Computing



Message Passing Interface (MPI)

- MPI is designed for distributed-memory parallelism, allowing processes to communicate and collaborate across multiple nodes in a cluster or supercomputer.
- Experiments related to:
 - Parallel Sum and
 - Parallel Vector Addition

Using Inter-Process Communication:

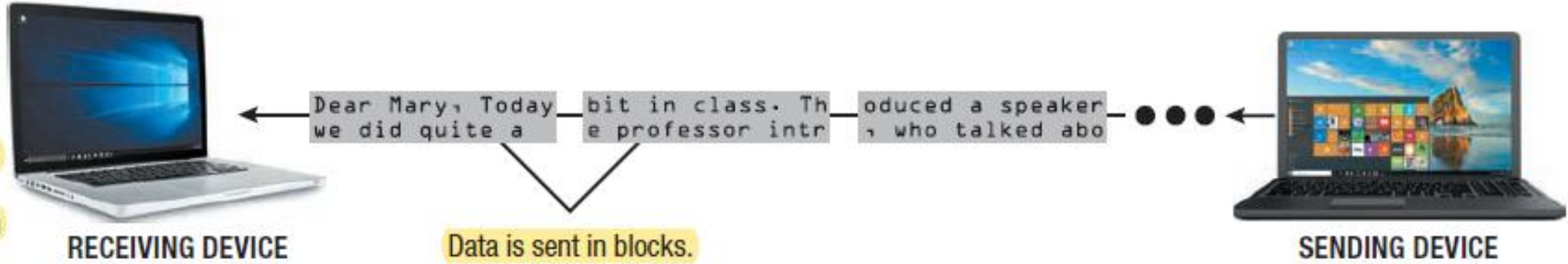
- Synchronous/Asynchronous
- Movement of data from one process's address space to another.

Synchronous vs Asynchronous

- A Synchronous communication is not complete until the message has been received.
- An Asynchronous communication is complete as soon as the message is on the way.

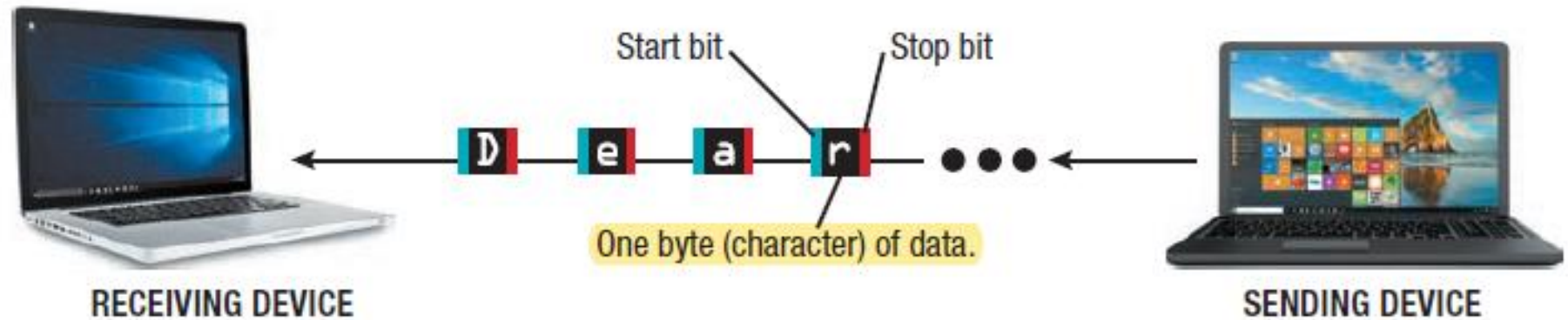
SYNCHRONOUS TRANSMISSIONS

Data is sent in blocks and the blocks are timed so that the receiving device knows when they will arrive.



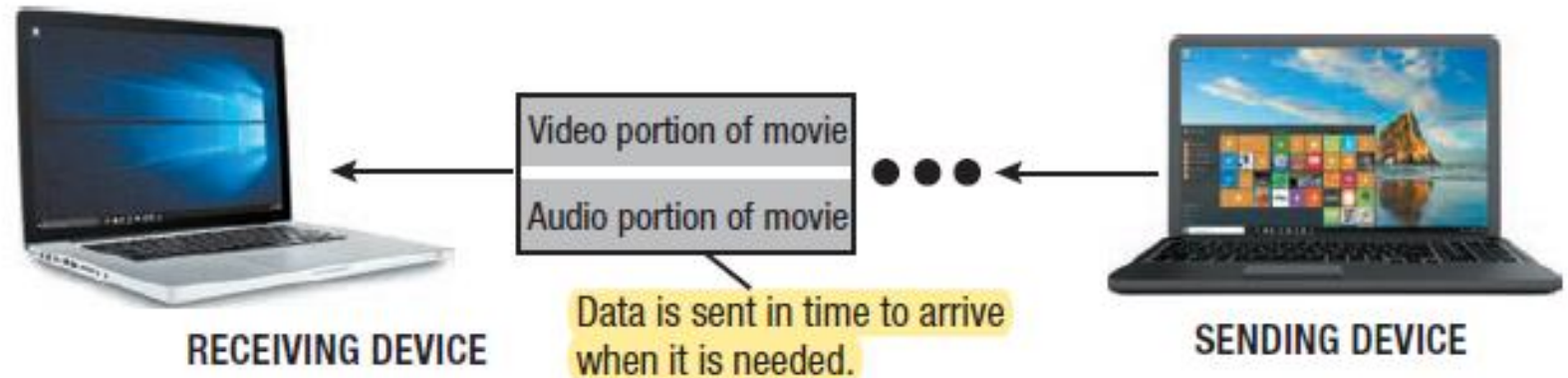
ASYNCHRONOUS TRANSMISSIONS

Data is sent one byte at a time, along with a start bit and a stop bit.



ISOCRONOUS TRANSMISSIONS

Data is sent when it is ready but only after requesting and being assigned the bandwidth necessary for all the data to arrive at the correct time.



Example 1

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

int main()
{

int thread;

#pragma omp parallel
{
thread = omp_get_thread_num();
printf("Thread %d: = hello world\n", thread);
}
}
```

```
Thread 2: = hello world
Thread 1: = hello world
Thread 0: = hello world
Thread 3: = hello world
```

- The term "pragma" is derived from "pragmatic information," as these directives offer a way for programmers to give guidance to the compiler about how to parallelize specific sections of code.

Example 2

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

int main()
{

int thread;

#pragma omp parallel num_threads (5)|
{
thread = omp_get_thread_num();
printf("Thread %d: = hello world\n", thread);
}
}
```

```
Thread 1: = hello world
Thread 3: = hello world
Thread 0: = hello world
Thread 2: = hello world
Thread 4: = hello world
```

```
Thread 3: = hello world
Thread 2: = hello world
Thread 0: = hello world
Thread 1: = hello world
Thread 4: = hello world
```

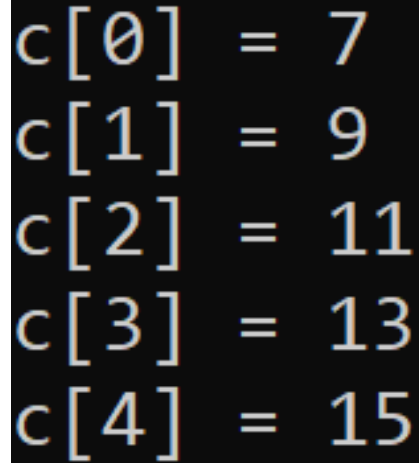
Simple Array Addition

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

int main()
{
    int a[5] = {1,2,3,4,5};
    int b[5] = {6,7,8,9,10};

    int c[5];
    int i;

    for(i=0; i<5; i++)
    {
        c[i] = a[i] + b[i];
        printf("c[%d] = %d\n",i,c[i]);
    }
}
```



c[0]	=	7
c[1]	=	9
c[2]	=	11
c[3]	=	13
c[4]	=	15

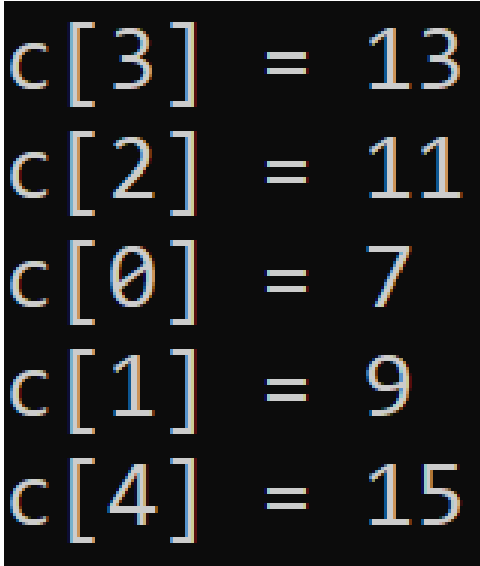
Array Addition with Threads

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

int main()
{
    int a[5] = {1,2,3,4,5};
    int b[5] = {6,7,8,9,10};

    int c[5];
    int thread;

    #pragma omp parallel num_threads(5)
    {
        thread = omp_get_thread_num();
        c[thread] = a[thread] + b[thread];
        printf("c[%d] = %d\n",thread,c[thread]);
    }
}
```



c[3] = 13
c[2] = 11
c[0] = 7
c[1] = 9
c[4] = 15

Array Addition with Threads more than the Array Size

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

int main()
{
int a[5] = {1,2,3,4,5};
int b[5] = {6,7,8,9,10};

int c[5];
int thread;

#pragma omp parallel num_threads(6)
{
thread = omp_get_thread_num();
c[thread] = a[thread] + b[thread];
printf("c[%d] = %d\n",thread,c[thread]);
}
}
```

```
c[1] = 9
c[2] = 11
c[0] = 7
c[4] = 15
c[3] = 13
c[5] = 6422265
```

Parallel 'for'

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

int main()
{

int thread;
int c[5];
int i;

#pragma omp parallel for num_threads(5)

for(i=0; i<5; i++)
{
thread = omp_get_thread_num();
printf("Thread %d : c[%d] = hello world\n", thread, i);
}
}
```

//each thread printing its own identification and accessing a specific index of the array.

```
Thread 4 : c[4] = hello world
Thread 1 : c[1] = hello world
Thread 2 : c[2] = hello world
Thread 0 : c[0] = hello world
Thread 3 : c[3] = hello world
```

‘nowait’

- **Without nowait**, threads would typically wait for all iterations of the loop to complete before executing the additional work. However,
- **With nowait**, the threads can proceed to the code after the loop immediately, potentially overlapping the execution of that code with the remaining iterations of the loop.

With 'nowait'

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

int main() {
    int thread;
    int c[5];
    int i;

    #pragma omp parallel num_threads(5)
    {
        #pragma omp for nowait
        for (i = 0; i < 5; i++) {
            thread = omp_get_thread_num();
            printf("Thread %d : c[%d] = hello world\n", thread, i);
        }

        // Additional work after the loop
        printf("Thread %d executing additional work after the loop\n", omp_get_thread_num());
    }

    // Code after the parallel region
    printf("Code after the parallel region\n");

    return 0;
}
```

With 'nowait'

```
Thread 0 : c[0] = hello world  
Thread 0 executing additional work after the loop  
Thread 1 : c[1] = hello world  
Thread 1 executing additional work after the loop  
Thread 4 : c[4] = hello world  
Thread 4 executing additional work after the loop  
Thread 2 : c[2] = hello world  
Thread 2 executing additional work after the loop  
Thread 3 : c[3] = hello world  
Thread 3 executing additional work after the loop  
Code after the parallel region
```

Without 'nowait'

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

int main() {
    int thread;
    int c[5];
    int i;

    #pragma omp parallel num_threads(5)
    {
        #pragma omp for
        for (i = 0; i < 5; i++) {
            thread = omp_get_thread_num();
            printf("Thread %d : c[%d] = hello world\n", thread, i);
        }

        // Additional work after the loop
        printf("Thread %d executing additional work after the loop\n", omp_get_thread_num());
    }

    // Code after the parallel region
    printf("Code after the parallel region\n");

    return 0;
}
```

Without 'nowait'

```
Thread 1 : c[1] = hello world
Thread 3 : c[3] = hello world
Thread 0 : c[0] = hello world
Thread 2 : c[2] = hello world
Thread 4 : c[4] = hello world
Thread 1 executing additional work after the loop
Thread 3 executing additional work after the loop
Thread 0 executing additional work after the loop
Thread 2 executing additional work after the loop
Thread 4 executing additional work after the loop
Code after the parallel region
```

'section'

- 'section' directive is used to specify that different sections of code can be executed in parallel by different threads.
- It doesn't fix threads for a particular work but rather provides a way to parallelize specific sections of code.
- The actual number of threads that execute the sections is determined by the runtime system.

'section'

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

```
int main() {
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            // Code for the first section, executed in parallel
            printf("Thread %d executing section 1\n", omp_get_thread_num());
        }

        #pragma omp section
        {
            // Code for the second section, executed in parallel
            printf("Thread %d executing section 2\n", omp_get_thread_num());
        }
    }

    return 0;
}
```

```
Thread 0 executing section 1
Thread 1 executing section 2
```

'critical' to avoid Data Races

- '**critical**' directive in OpenMP is used to specify a region of code that must be executed by only one thread at a time.
- This can be useful when you have a section of code that should not be executed concurrently by multiple threads to avoid data races or other issues.

'critical'

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

int main() {
    // Shared variable
    int sharedVariable = 0;

    // Parallel region with a critical section
    #pragma omp parallel num_threads(4)
    {
        int threadID = omp_get_thread_num();

        #pragma omp critical
        {
            // Only one thread can execute this section at a time
            printf("Thread %d is in the critical section\n", threadID);
            sharedVariable += 1;
        }

        //Use 'barrier' to synchronize threads before moving on to next code
        #pragma omp barrier
        {
            printf("Thread %d is outside the critical section\n", threadID);
        }
    }

    // Print the final value of the shared variable
    printf("Final value of sharedVariable: %d\n", sharedVariable);

    return 0;
}
```

‘critical’

```
Thread 0 is in the critical section  
Thread 1 is in the critical section  
Thread 2 is in the critical section  
Thread 3 is in the critical section  
Thread 0 is outside the critical section  
Thread 2 is outside the critical section  
Thread 3 is outside the critical section  
Thread 1 is outside the critical section  
Final value of sharedVariable: 4
```

'barrier'

- '**barrier**' directive ensures that all threads execute the code before the barrier before moving on to the code after the barrier.
- All threads synchronize at the barrier.