



# Parallel and Distributed Computing

## CS3006

Lecture 8

**Shared Memory & OpenMP**

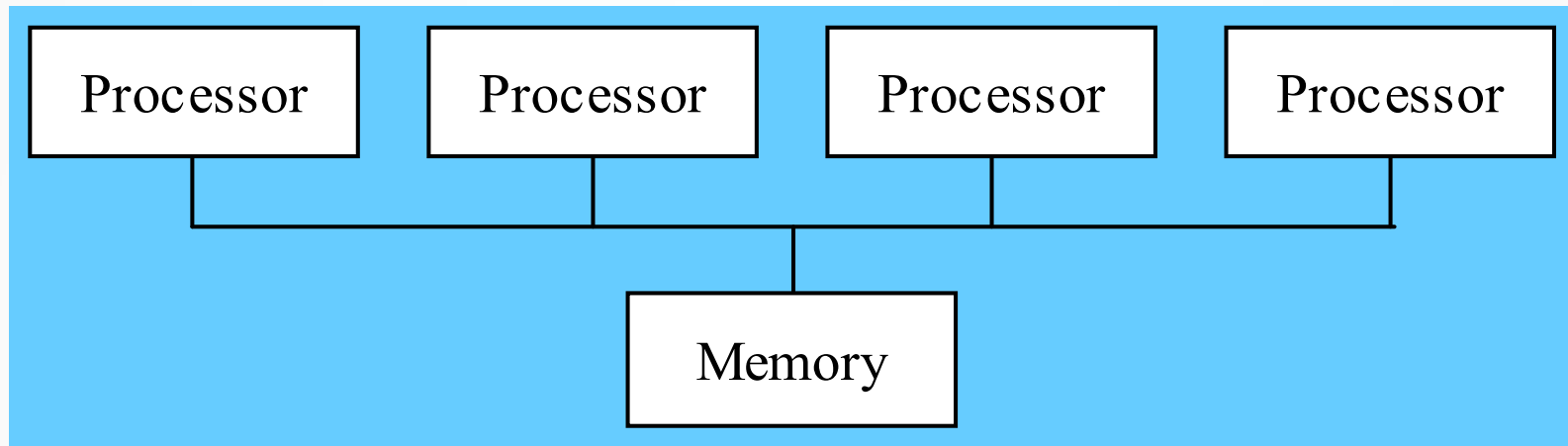
4th March 2024

Dr. Rana Asif Rehman

# Shared-Memory Programming

- **Physically:** processors in a computer share access to the same RAM
- **Virtually:** threads running on the processors interact with one another via shared variables in the common address space of a single process
- Making performance improvements to serial code tends to be **easier** with multithreading than with message passing paradigm
  - Message passing usually requires the code/algorithm to be redesigned
  - Multithreading allows incremental parallelism: take it one loop at a time
- Clusters today are commonly made up of multiple processors per compute node; using OpenMP with MPI is a strategy to improve performance at the two levels (i.e., shared and distributed memory)

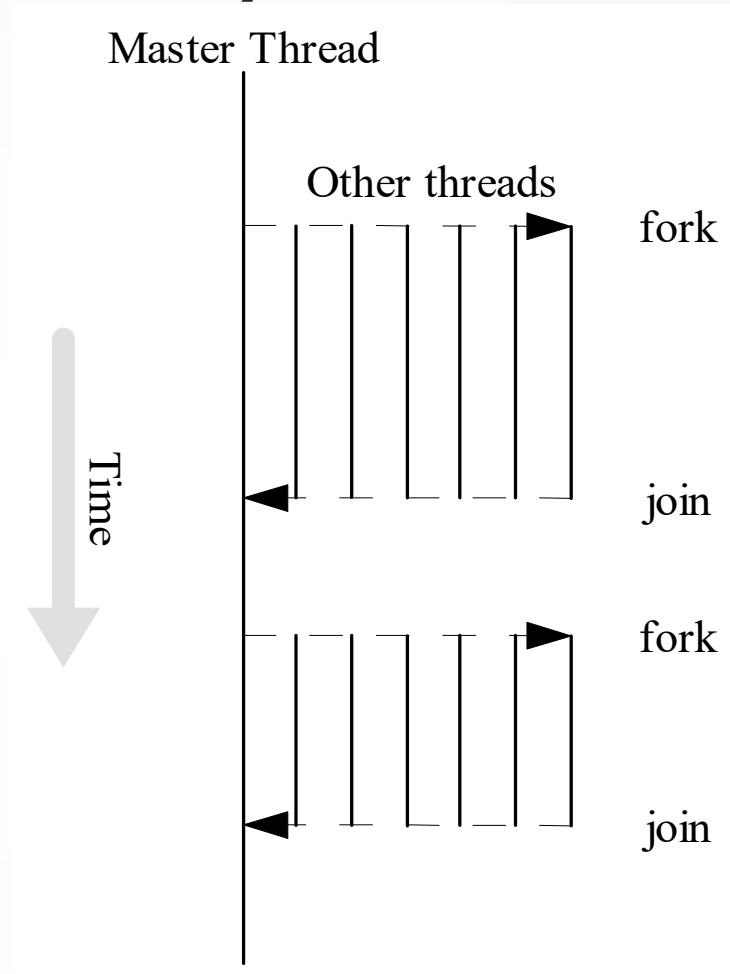
# Shared-Memory Programming



Processors interact and synchronize with each other through shared variables.

# Shared-Memory Programming (Fork/Join Parallelism)

- Multi-threaded programming is the most common shared-memory programming methodology.
- A serial code begins execution. The process is the master thread or only executing thread.
- When a parallel portion of the code is reached, the Master thread can “fork” more threads to work on it.
- When the parallel portion of the code has completed, the threads “join” again, and the master thread continues executing the serial code



# Shared-Memory vs. Message Passing Model

## ➤ Shared-memory model

- One active thread at start and end of the program
- Number of active threads inside program changes dynamically during execution.
- Supports incremental parallelism
  - the process of converting a sequential program to a parallel program a little bit at a time

## ➤ Message-passing model

- All processes remain active throughout execution of program
- Sequential-to-parallel transformation requires major effort
- No incremental parallelism
  - Transformation done in one giant step rather than many tiny steps

# Introduction to OpenMP

- OpenMP has emerged as a standard method for shared-memory programming
  - Same as MPI has become the standard for distributed-memory programming
  - Codes are portable
  - Performance is usually good enough
- Consists of compiler directives, Library functions, and environment variables
- Compiler support
  - C, C++ & Fortran
  - Intel (icc -openmp), and GNU (gcc -fopenmp)

# How does it work?

- C programs often express data-parallel operations as **for** loops

```
for (i = first; i < size; i += 2)
    marked[i] = 1;
```

- OpenMP makes it easy to indicate when the iterations of a loop may execute in parallel
- Compiler takes care of generating code that forks/joins threads and allocates the iterations to threads



# How does it work ? (Pragmas)

- **Pragma:** a compiler directive in C or C++
  - Stands for “pragmatic information”
  - A way for the programmer to communicate with the compiler
  - Compiler is free to ignore pragmas
- Syntax:  
**#pragma omp** *<rest of pragma>*
- The pragmas precede the regions that can be parallelize to flag the compiler that performing operations in parallel does not affect the program semantics (i.e., doesn't affect the program's logic).





# How does it work? (parallel for Pragma)

➤ Format:

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

- Compiler must be able to verify **total number of iteration** before executing the program, to parallelize it
- **Body of for-loop** must not allow premature exits (e.g., *break*, *return*, *exit*, or *goto* statements are not allowed).
- However, loops with 'continue' statement are allowed as it does not cause the premature exit.

# Canonical [allowed] Shape of *for-loop* Condition

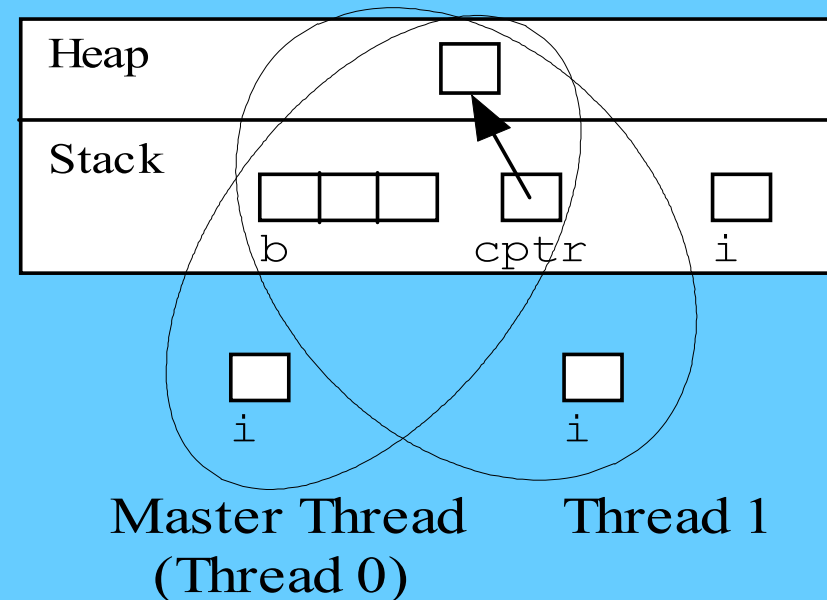
► Here 'inc' can be any constant value

$$\text{for (index} = \textit{start} ; \text{index} \left\{ \begin{array}{l} < \\ \leq \\ \geq \\ > \end{array} \right\} \textit{end} ; \left\{ \begin{array}{l} \text{index} \text{ } ++ \\ ++\text{index} \\ \text{index} \text{ } -- \\ --\text{index} \\ \text{index} \text{ } += \textit{inc} \\ \text{index} \text{ } -= \textit{inc} \\ \text{index} = \text{index} + \textit{inc} \\ \text{index} = \textit{inc} + \text{index} \\ \text{index} = \text{index} - \textit{inc} \end{array} \right\} )$$

# Shared and Private Variables

- Shared variable: has same address in execution context of every thread
- Private variable: has different address in execution context of every thread
- A thread cannot access the private variables of another thread

```
int main (int argc, char *argv[])  
{  
    int b[3];  
    char *cptr;  
    int i;  
  
    cptr = malloc(1);  
    #pragma omp parallel for  
    for (i = 0; i < 3; i++)  
        b[i] = i;  
}
```



# Basic Library Functions



## ➤ **omp\_get\_num\_procs**

```
int procs = omp_get_num_procs() //number of CPUs/cores in machine
```

## ➤ **omp\_get\_num\_threads**

```
int threads = omp_get_num_threads() //# of active threads  
//should be called from a parallel region
```

## ➤ **omp\_get\_max\_threads**

```
printf("Only %d threads can be forked\n",omp_get_max_threads());  
//can be called outside of a parallel region. Returns value of env. //variable  
'OMP_NUM_THREADS'
```


## ➤ **omp\_get\_thread\_num**

```
printf("Hello from thread id %d\n",omp_get_thread_num());
```

## ➤ **omp\_set\_num\_threads** –

```
omp_set_dynamic(0); // disable dynamic adjustment  
omp_set_num_threads(4); //setting thread count to 4
```

# helloworld.c



```
int main(){
    int procs,i,a,b,c,t,m;

    // Determine the number of physical processors
    procs = omp_get_num_procs();
    printf("%d processors/cores\n",procs);

    /* Determine max threads defined by default
       through the OMP_NUM_THREADS environment variable */
    printf("OMP_NUM_THREADS = %d\n",omp_get_max_threads());

    #pragma omp parallel
    printf("Hello from thread id %d\n",omp_get_thread_num());
}
```

# Questions



Parallel and Distributed Computing  
(CS3006) - Spring 2024



# References

1. Kumar, V., Grama, A., Gupta, A., & Karypis, G. (2017). *Introduction to parallel computing*. Redwood City, CA: Benjamin/Cummings.