सुस्वागतम्
நல்வரவு
ಸುಸ್ವಾಗತಂ
సుస్వాగతం
સુસ્વાગતમ
সুস্বাগতম
ସୁସ୍ୱାଗତମ
സുസ്വാഗതം
ਸੁਆਗਤਮ
सुस्वागतम्
خوش آمدید

Ministry of Electronics and Information Technology
Government of India

Introduction to Parallel Programming

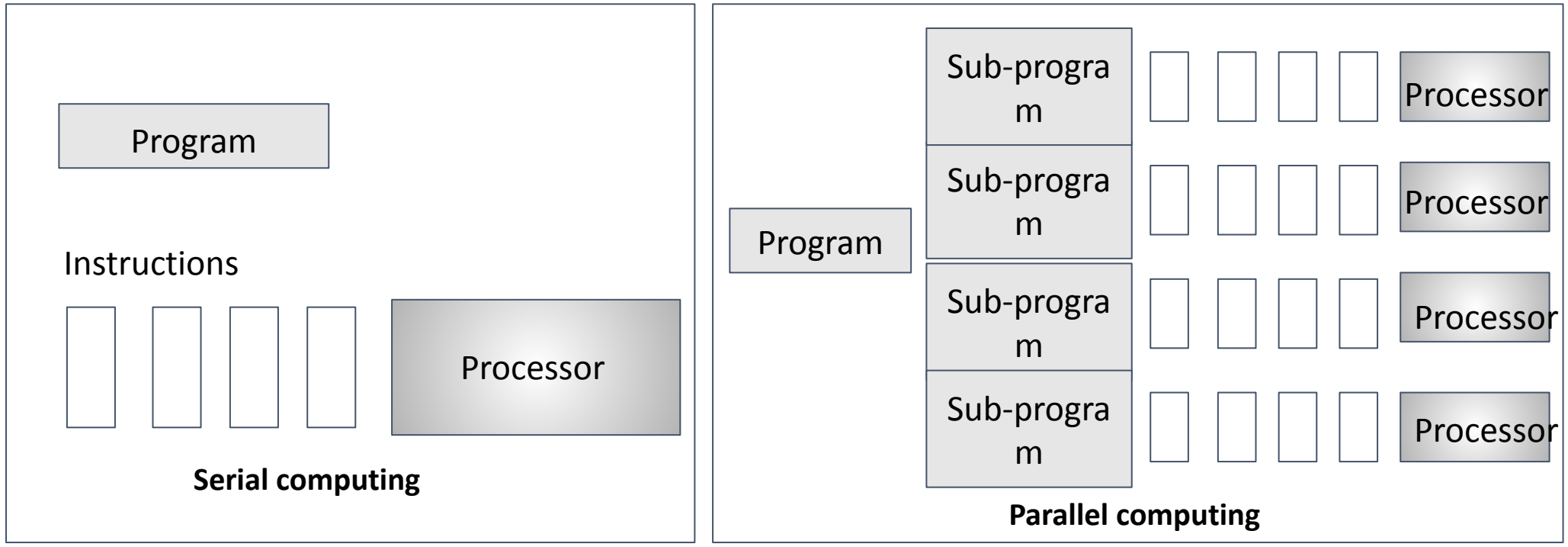
Harsha Ugave
HPC-Tech
CDAC Pune

Content

- Introduction to parallel programming
  - What is Parallel Programming?
  - Need of Parallel programming.
  - Why Parallel programming?
- Introduction to parallel hardware.
- General Parallel Computing Terminology
- Process and Threads.
- Demo

Introduction to parallel programming

What is parallel programming?



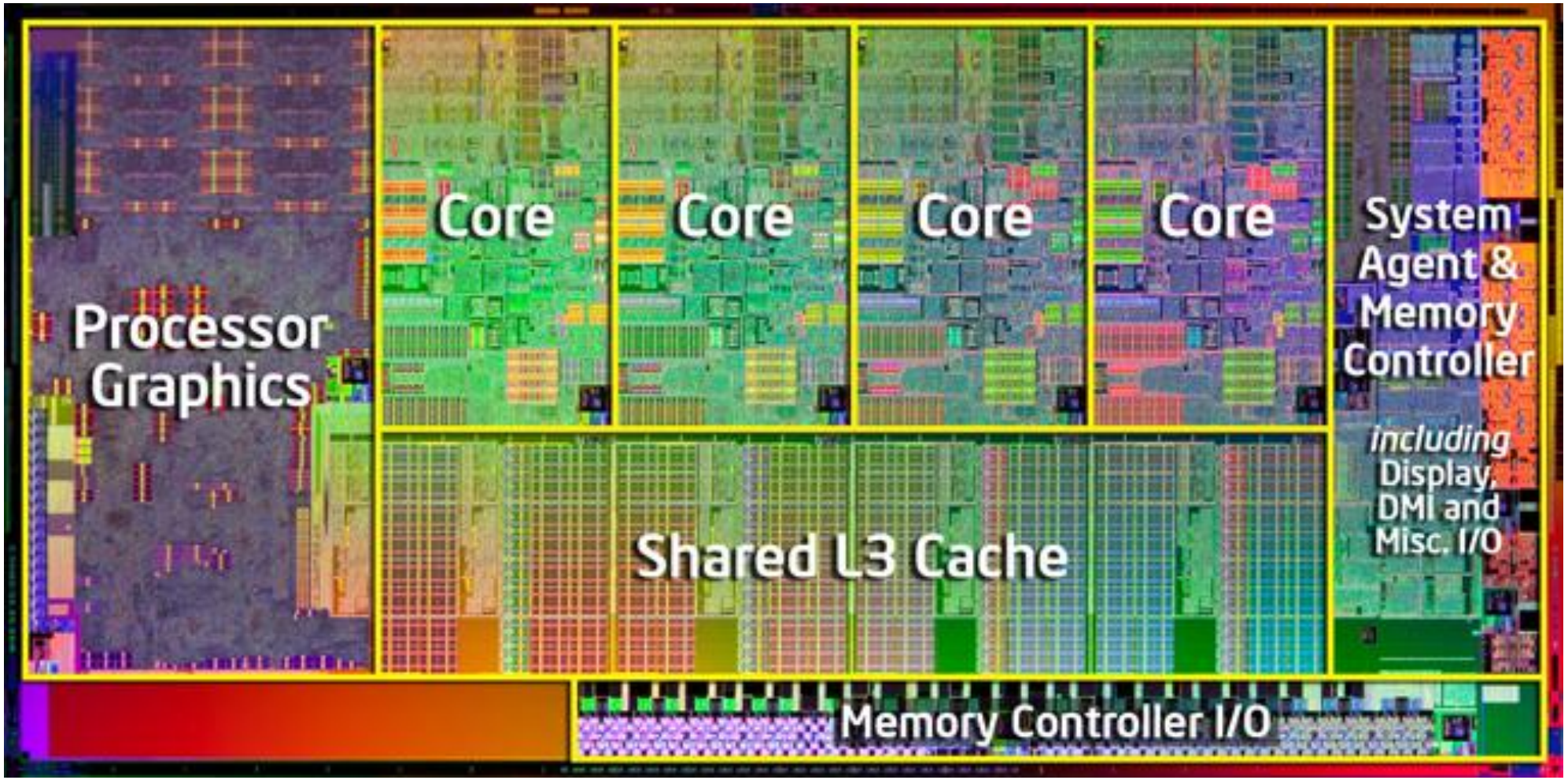
- Need of parallel programming.
  - The exponential growth of processing and network speeds means that parallel architecture isn’t just a good idea; it’s necessary. Big data and the IoT will soon force us to crunch trillions of data points at once.
  - Dual-core, quad-core, 8-core, and even 56-core chips are all examples of parallel computing. So, while parallel computers aren’t new, here’s the rub: new technologies are cranking out ever-faster networks, and computer performance has grown 250,000 times in 20 years.
  - For instance, in just the healthcare sector, AI tools will be rifling through the heart rates of a hundred million patients, looking for the telltale signs of A-fib or V-tach and saving lives. They won’t be able to make it work if they have to plod along performing one operation at a time

Introduction to parallel programming

Why parallel programming?

- PARALLEL COMPUTING MODELS THE REAL WORLD.
- SAVES TIME
- SAVES MONEY
- SOLVE MORE COMPLEX OR LARGER PROBLEMS
- MAKE BETTER USE OF UNDERLYING PARALLEL HARDWARE

Introduction to parallel hardware.



- Node

  A standalone "computer in a box." Usually comprised of multiple CPUs/processors/cores, memory, network interfaces, etc. Nodes are networked together to comprise a supercomputer.

- CPU

  Contemporary CPUs consist of one or more cores.

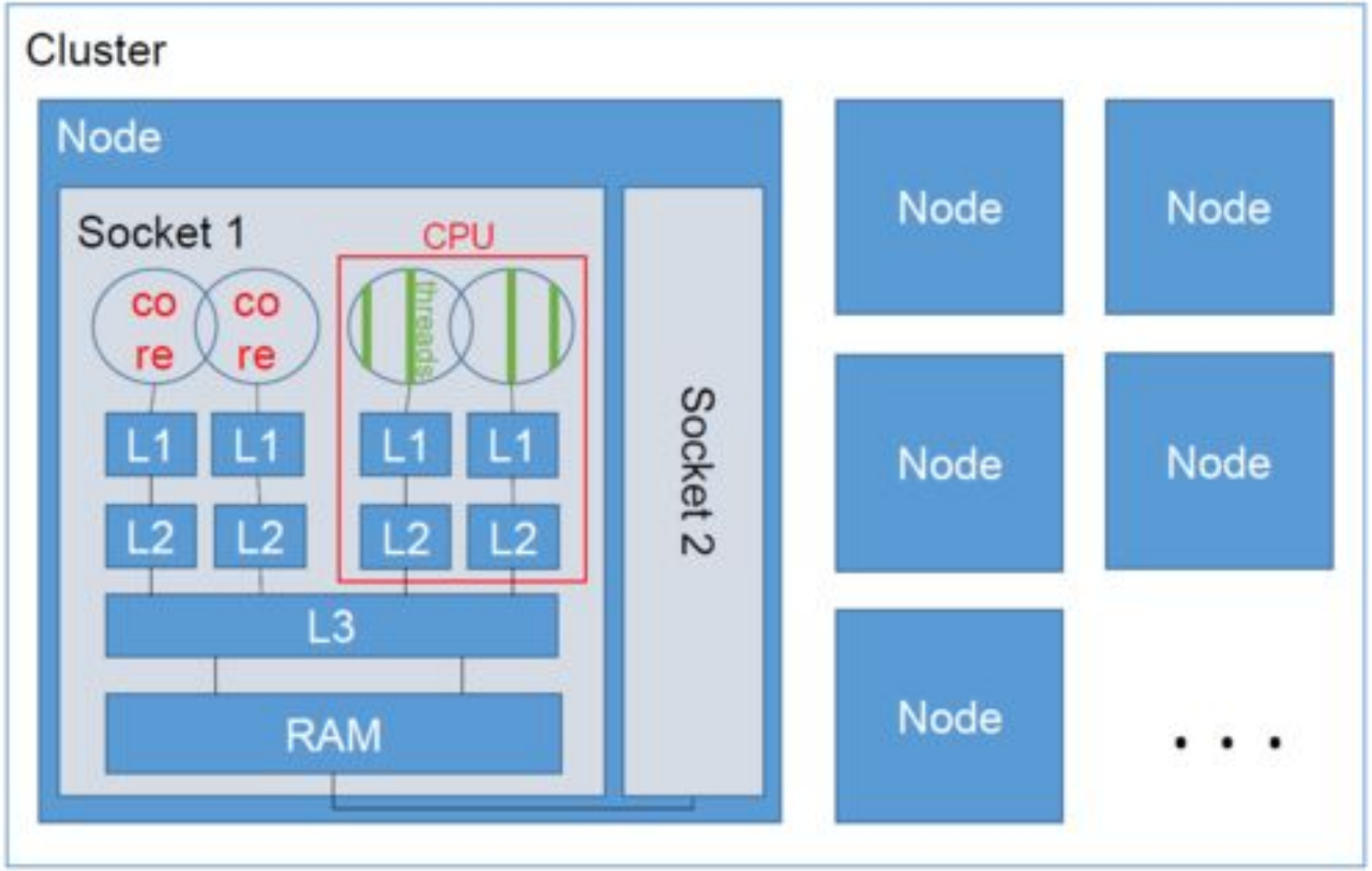
- Process

  A process is an instance of a program that is being executed or processed

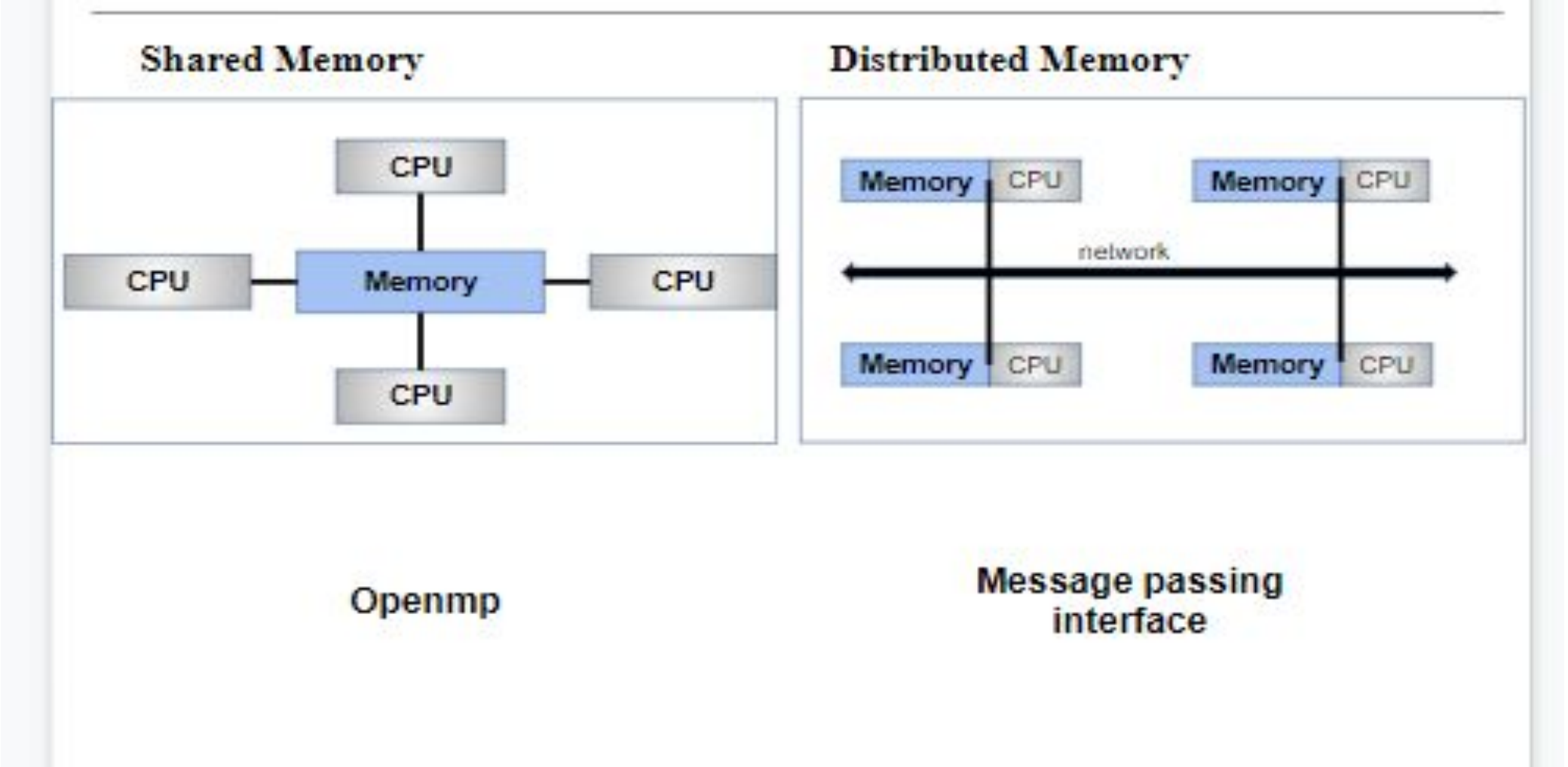
- Thread

  Thread is a segment of a process or a lightweight process that is managed by the scheduler independently

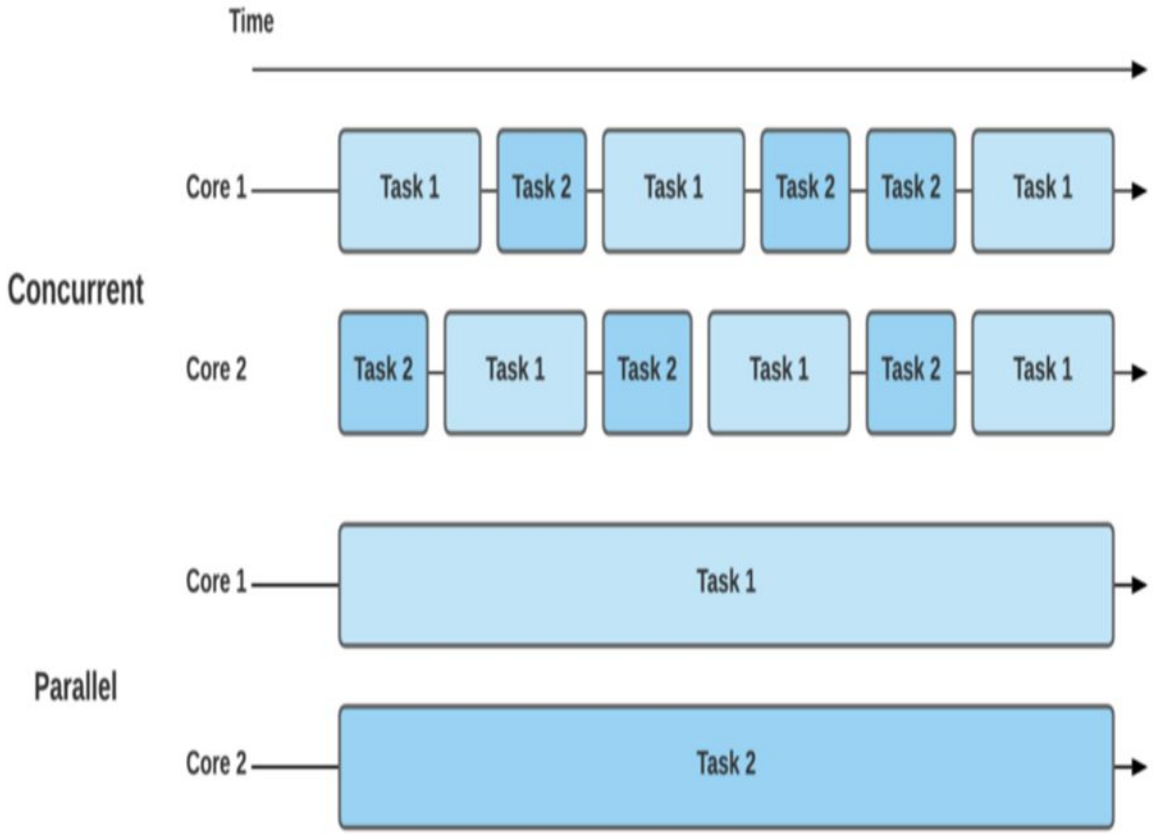
General Parallel Computing Terminology



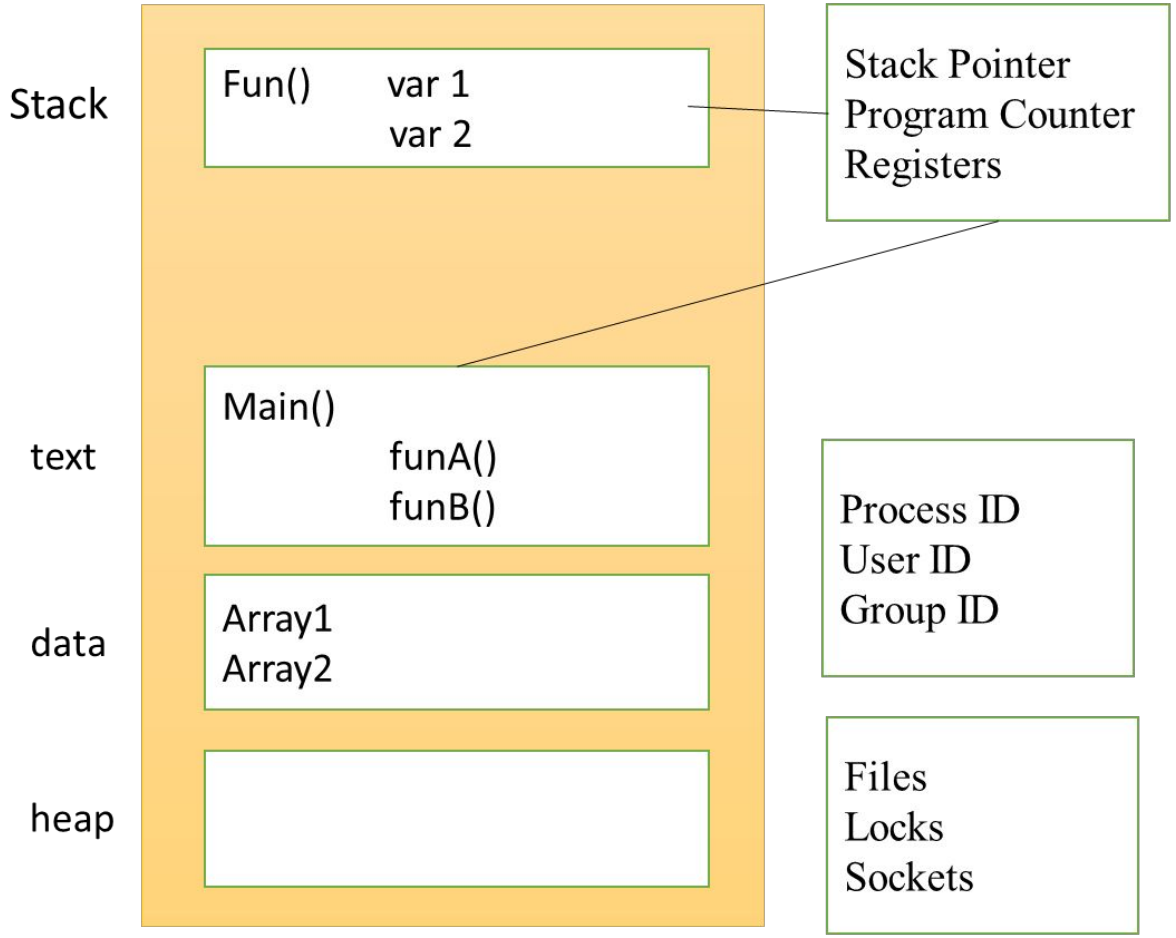
General Parallel Computing Terminology



- Concurrency and Parallelism
  - Concurrency: A condition of a system in which multiple tasks are logically active at one time
  - Parallelism: A condition of a system in which multiple tasks are actually active at one time



- How program execute in memory?

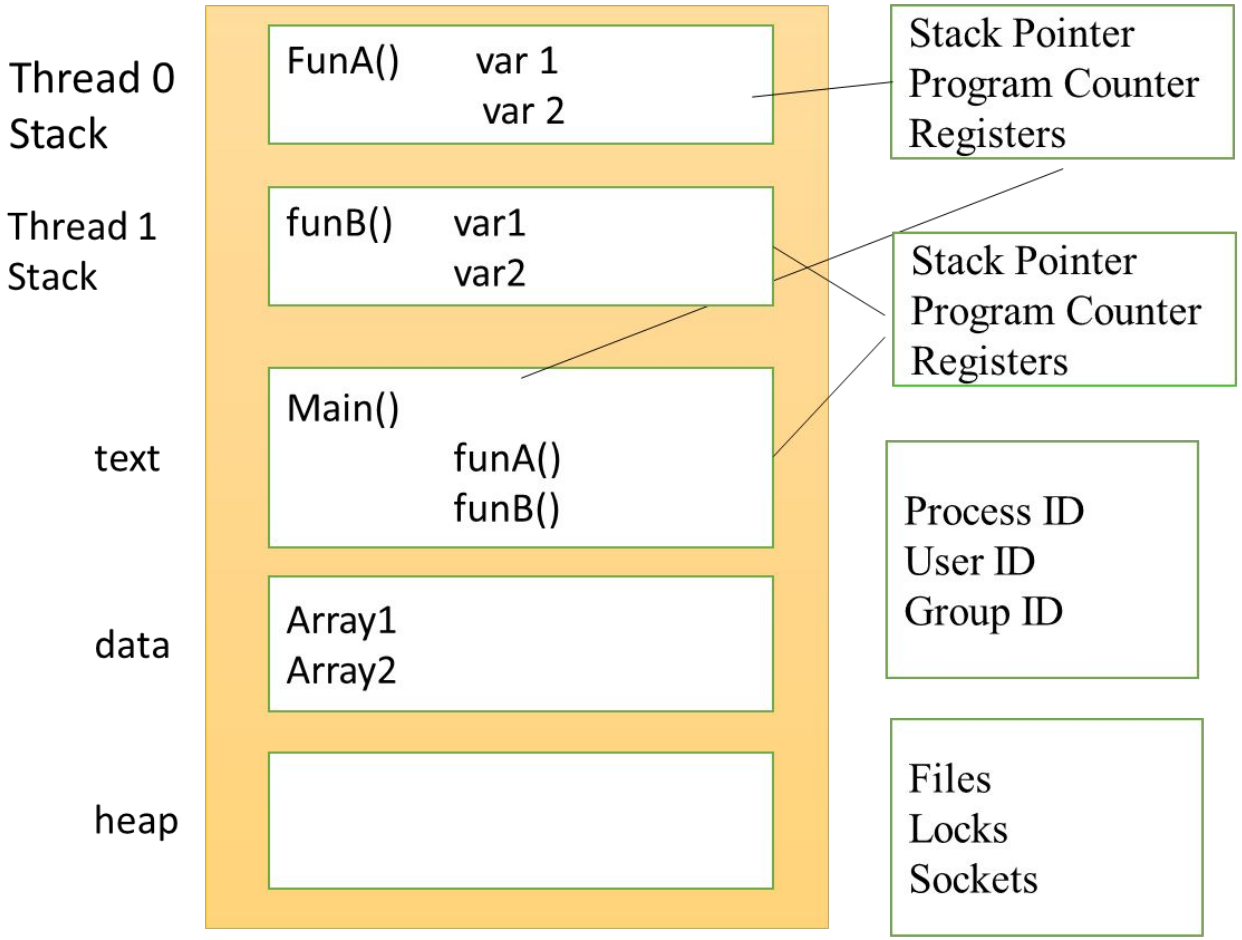


Process:

- An instance of a program execution.

- The execution context of a running program … i.e. the resources associated with a program's execution.

- How program execute in shared memory?



Threads:

- Threads are "light weight processes"

- Threads share Process state among multiple threads … this greatly reduces the cost of switching context.

Demo Time

Hardware information

$lscpu or cat /proc/cpuinfo

```
(base) [cdacapp@login03 ~]$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                40
On-line CPU(s) list:   0-39
Thread(s) per core:    1
Core(s) per socket:    20
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 85
Model name:            Intel(R) Xeon(R) Gold 6248 CPU @ 2.50GHz
Stepping:              7
CPU MHz:               999.908
CPU max MHz:           3900.0000
CPU min MHz:           1000.0000
BogoMIPS:              5000.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              1024K
L3 cache:              28160K
NUMA node0 CPU(s):     0-19
NUMA node1 CPU(s):     20-39
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts ac
pi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts rep_goo
d nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sd
bg fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand
 lahf_lm abm 3dnowprefetch epb cat_l3 cdp_l3 invpcid_single intel_ppin intel_pt ssbd mba ibrs ibpb stibp ibrs
_enhanced tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm
cqm mpx rdt_a avx512f avx512dq rdseed adx smap clflushopt clwb avx512cd avx512bw avx512vl xsaveopt xsavec xge
tbv1 cqm_llc cqm_occup_llc cqm_mbm_total cqm_mbm_local dtherm ida arat pln pts pku ospke avx512_vnni md_clear
 spec_ctrl intel_stibp flush_l1d arch_capabilities
(base) [cdacapp@login03 ~]$
```

Process and Thread information

$top –u <username>

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
43243	cdacapp	20	0	167096	3204	1664	R	1.0	0.0	0:00.09	top
28610	cdacapp	20	0	6498508	296596	1560	S	0.0	0.1	0:09.44	julia
28611	cdacapp	20	0	6493920	300736	1564	S	0.0	0.1	0:08.59	julia
28612	cdacapp	20	0	6569328	327268	1592	S	0.0	0.1	0:08.79	julia
28613	cdacapp	20	0	6565220	330992	1568	S	0.0	0.1	0:09.04	julia
42747	cdacapp	20	0	167760	2724	1208	S	0.0	0.0	0:00.04	sshd
42748	cdacapp	20	0	121768	4188	1728	S	0.0	0.0	0:00.18	bash

Program in C

```
#include <stdio.h>
int main() {
    int n, i, sum = 0;

    printf("Enter a positive integer: ");
    scanf("%d", &n);

    for (i = 1; i <= n; ++i) {
        sum += i;
    }

    printf("Sum = %d", sum);
    return 0;
}
```

How to compile:

$gcc <file_name>.c –o <file_name>

$time <file_name>

An Operating system’s perspective

- Supports multi-core
- Perceives each core as a separate processor
- Scheduler maps threds/process to different cores

Multi-core CPU

- Multi-core processor are MIMD where different cores execute different threads (MI), operating on different part of memory (Multiple data).

- Multi-core is a shared memory multiprocessor (SMP) where all cores share memory

Assignment

- What is FLOPS
- List of compilers we use in HPC
- Study Shared memory architecture
-

Thank you

Introduction to OPENMP

Open Multi Processing

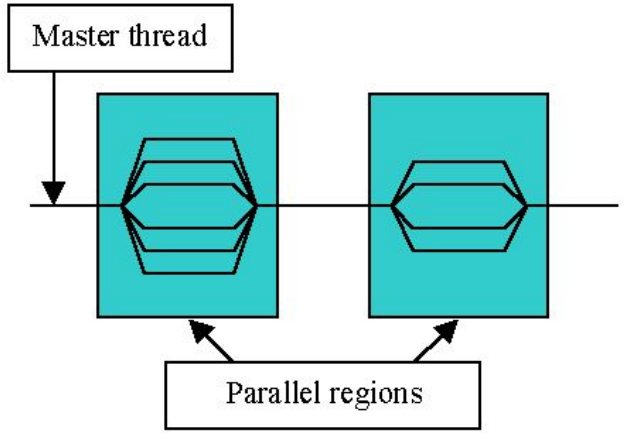
Content

- Introduction to openmp
  - What is OpenMP?
  - History of OpenMP
  - Why openmp
- OpenMP Programming Model
- OpenMP Stack
- Hello world in openmp
  - Basic Syntax
  - Hello world c program
  - Compile and execution

Introduction to openmp

What is OpenMP?

- An Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism
- Comprises three primary API components
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables



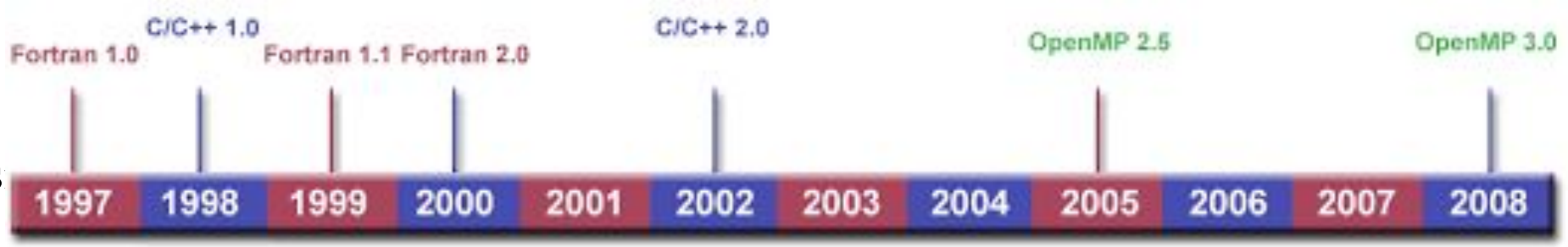
- Portable
  - The API is specified for C/C++ and Fortran
  - Has been implemented for most major platforms including Unix/ Linux platforms and Windows NT

What is OpenMP? (cont.)

- Standardized
  - Jointly defined and endorsed by a group of major computer hardware and software vendors
  - Expected to become an ANSI standard later???
- What does OpenMP stand for?
  - Short version: Open Multi-Processing
  - Long version: Open specifications for Multi-Processing via collaborative work between interested parties from the hardware and software industry, government and academia

Introduction to openmp

History of OpenMP



Vers

Version 2.0 : OpenMP for Fortran 1.1, in 1999

Version 2.0 : OpenMP for C/C++, in 2000

Version 2.0 : OpenMP for C/C++, in 2002

Version 2.5 : OpenMP for C/C++/Fortran, in 2005

History of OpenMP(cont.)

- Up to version 2.0, OpenMP primarily specified ways to parallelize highly regular loops, as they occur in matrix-oriented numerical programming,

- In Version 3.0 Included features like concept of tasks and the task construct.

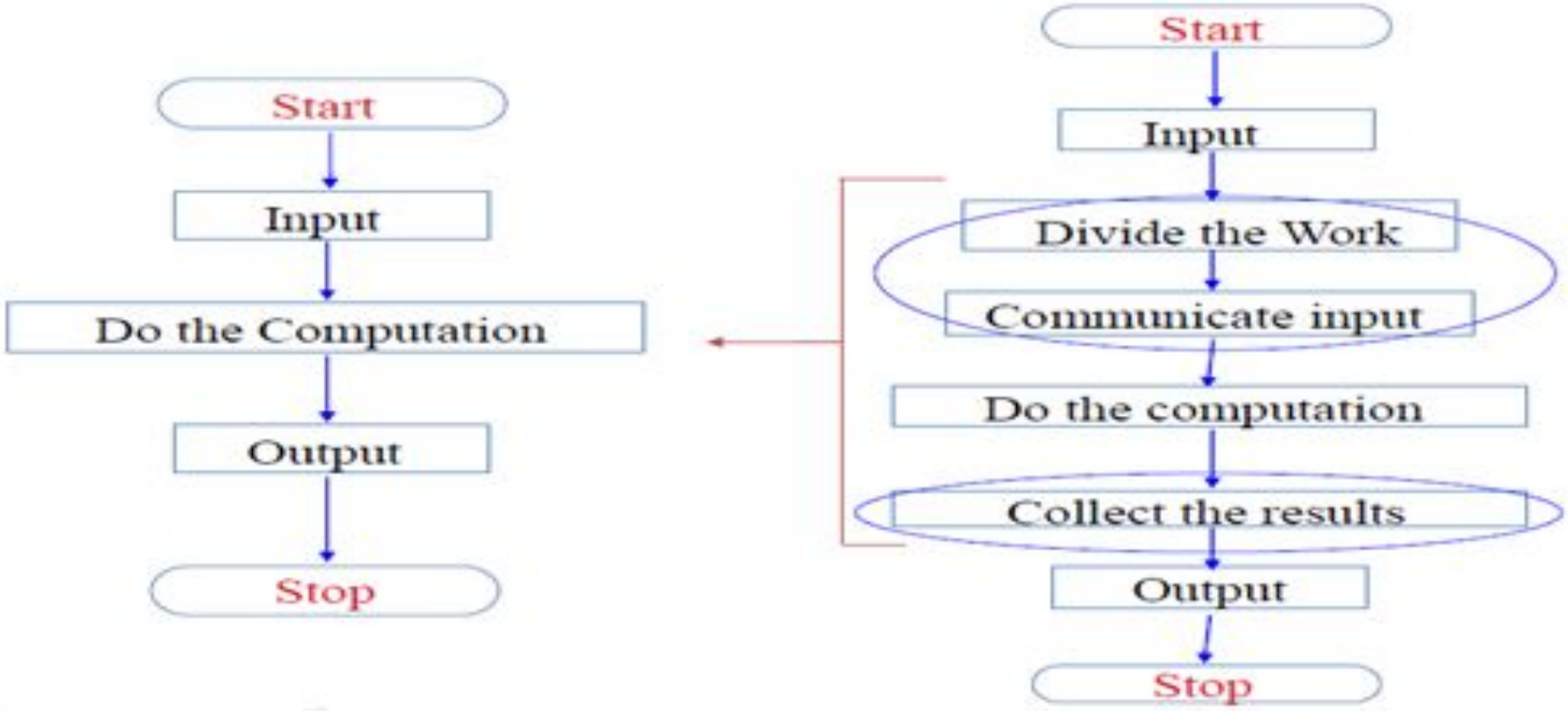
- In version 4.0 openmp improved following features:
- support for accelerators, atomics, error handling, thread affinity; tasking extensions; user defined reduction, SIMD support.

Introduction to openmp

Why Openmp?

- More efficient
- Hides the low-level details.
- OpenMP has directives that allow the programmer to:
  - specify the parallel region
  - specify whether the variables in the parallel section are private or shared
  - specify how/if the threads are synchronized
  - specify how to parallelize loops
  - specify how the works is divided between threads (scheduling)

Execution flow of Parallel Systems

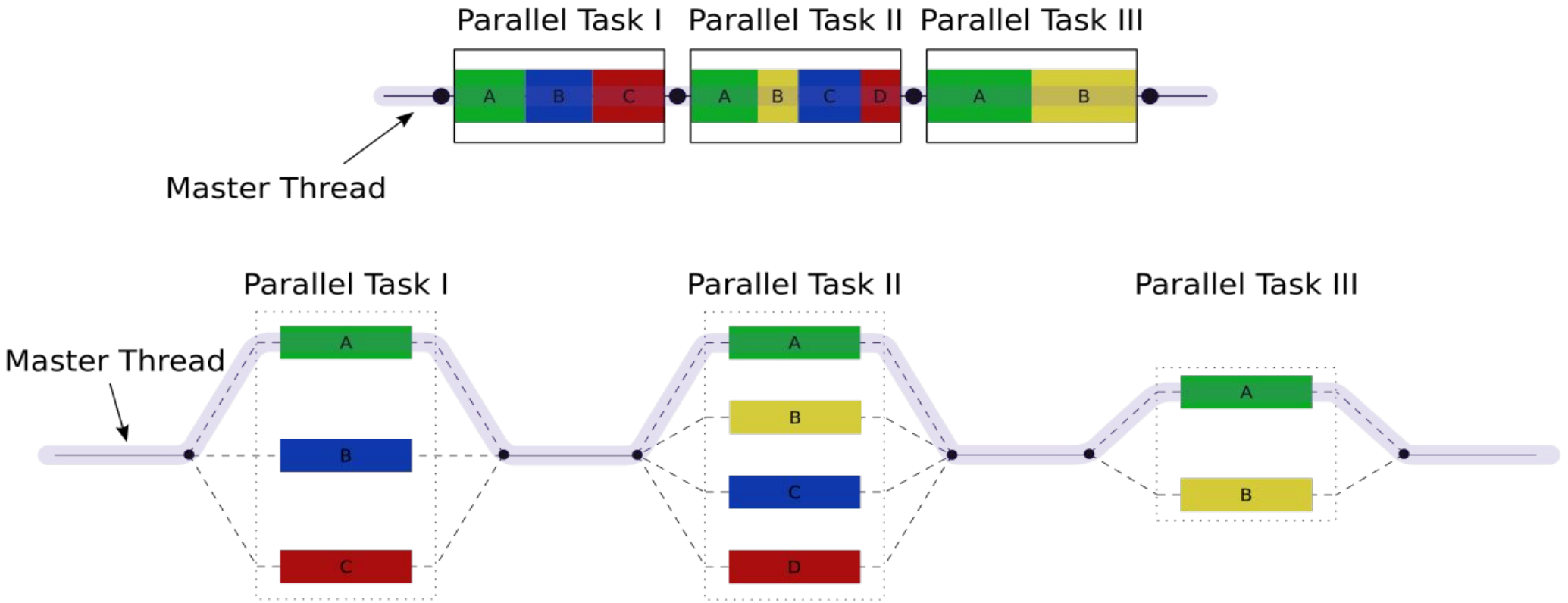


OpenMP Programming Model

- Shared memory, thread-based parallelism
  - OpenMP is based upon the existence of multiple threads in the shared memory programming paradigm.
  - A shared memory process consists of multiple threads.
- Explicit Parallelism
  - OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization.
- OpenMP uses the fork-join model of parallel execution.
- Compiler directive based
  - Most OpenMP parallelism is specified through the use of compiler directives which are imbedded in C/C++ or Fortran source code.

OpenMP Programming Model

Fork–join model:

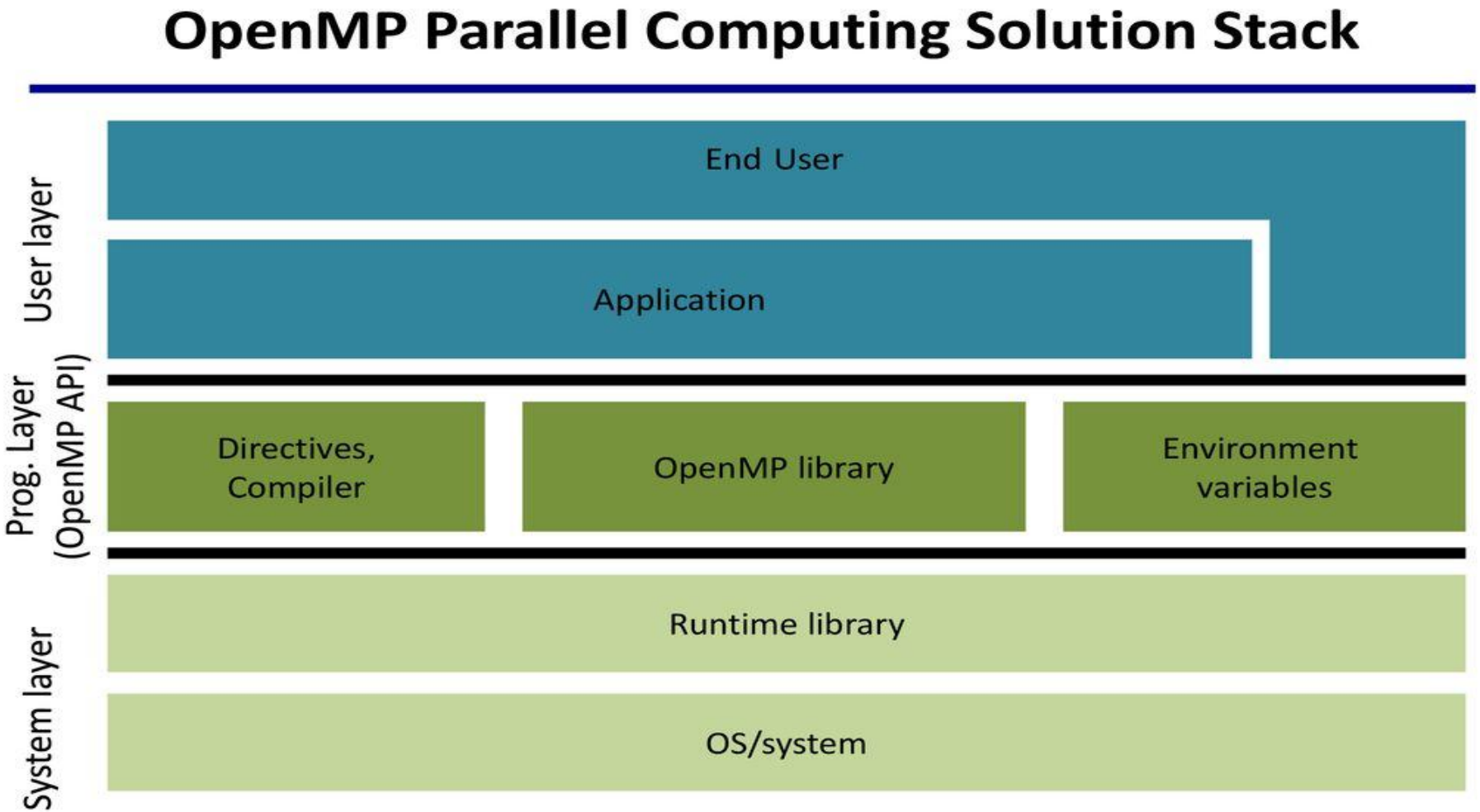


OpenMP Programming Model

Fork–join model: (cont.)

- All OpenMP programs begin as a single process: the master thread. The **master thread** executes sequentially until the first parallel region construct is encountered.
- **FORK**: the master thread then creates a **team** of parallel threads
- The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the team threads.
- **JOIN**: When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread

OpenMP Stack



Hello world in openmp

Basic Syntax

- Function prototypes and types in the file:

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

- Most of the constructs in OpenMP are compiler directives.

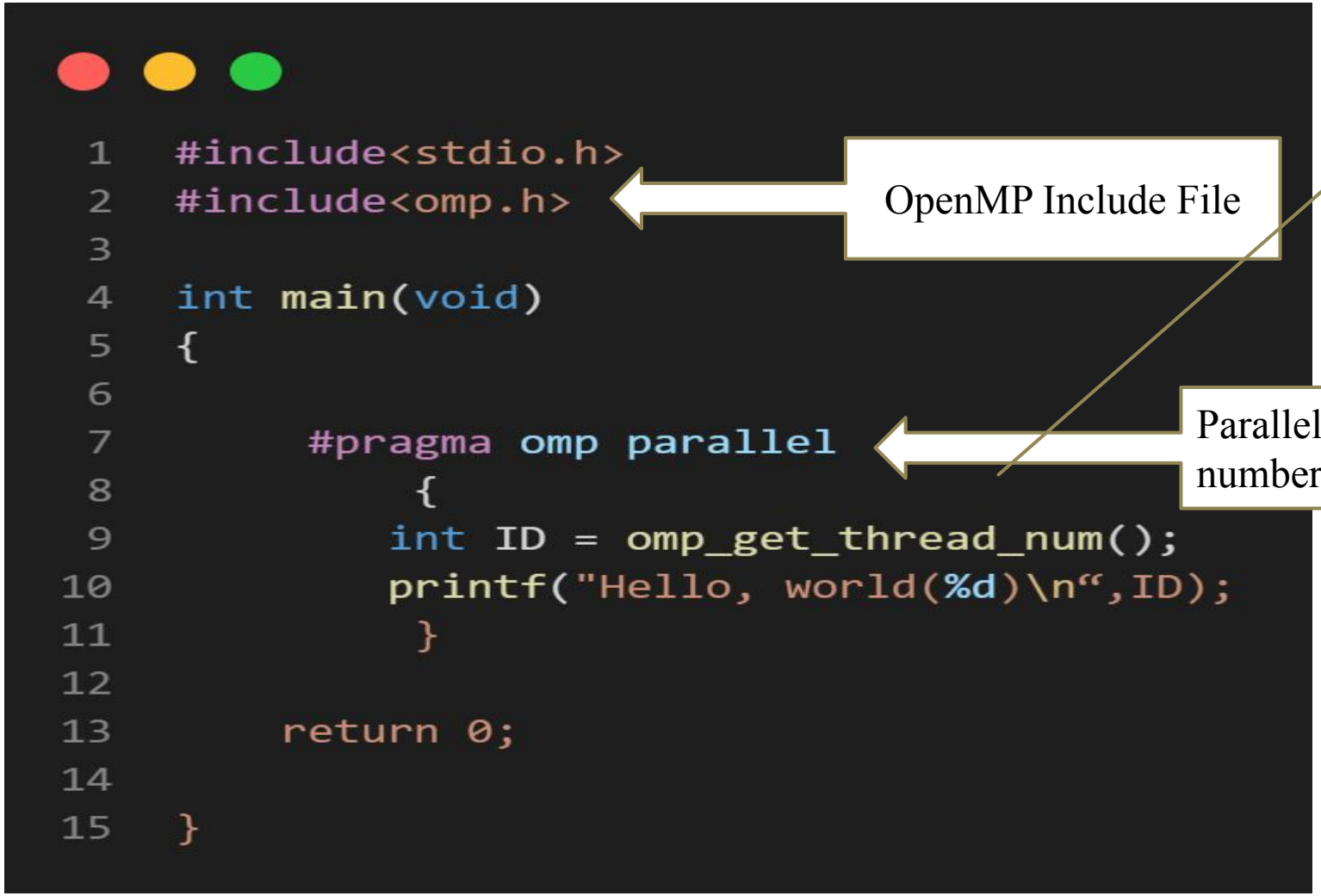
```
#pragma omp construct [clause [clause]…]
{
//..Do some work here
}
//end of parallel region/block
```

- Example:

```
#pragma omp parallel num_threads(4)
```

Hello world in openmp

OpenMP Hello World program using C



Your first openMP program

Check your system support: locate omp.h

Compilation: g++ -fopenmp hello.c

Execution: ./a.out

Flags:

GNU: -fopenmp for Linux, Solaris, AIX, MacOS, Windows.

IBM: -qsmp=omp for windows, AIX and Linux.

Sun: -xopwnmp for Solaris and Linux

Intel: -opwnmp on Linux or Mac, or -Qopenmp on windows

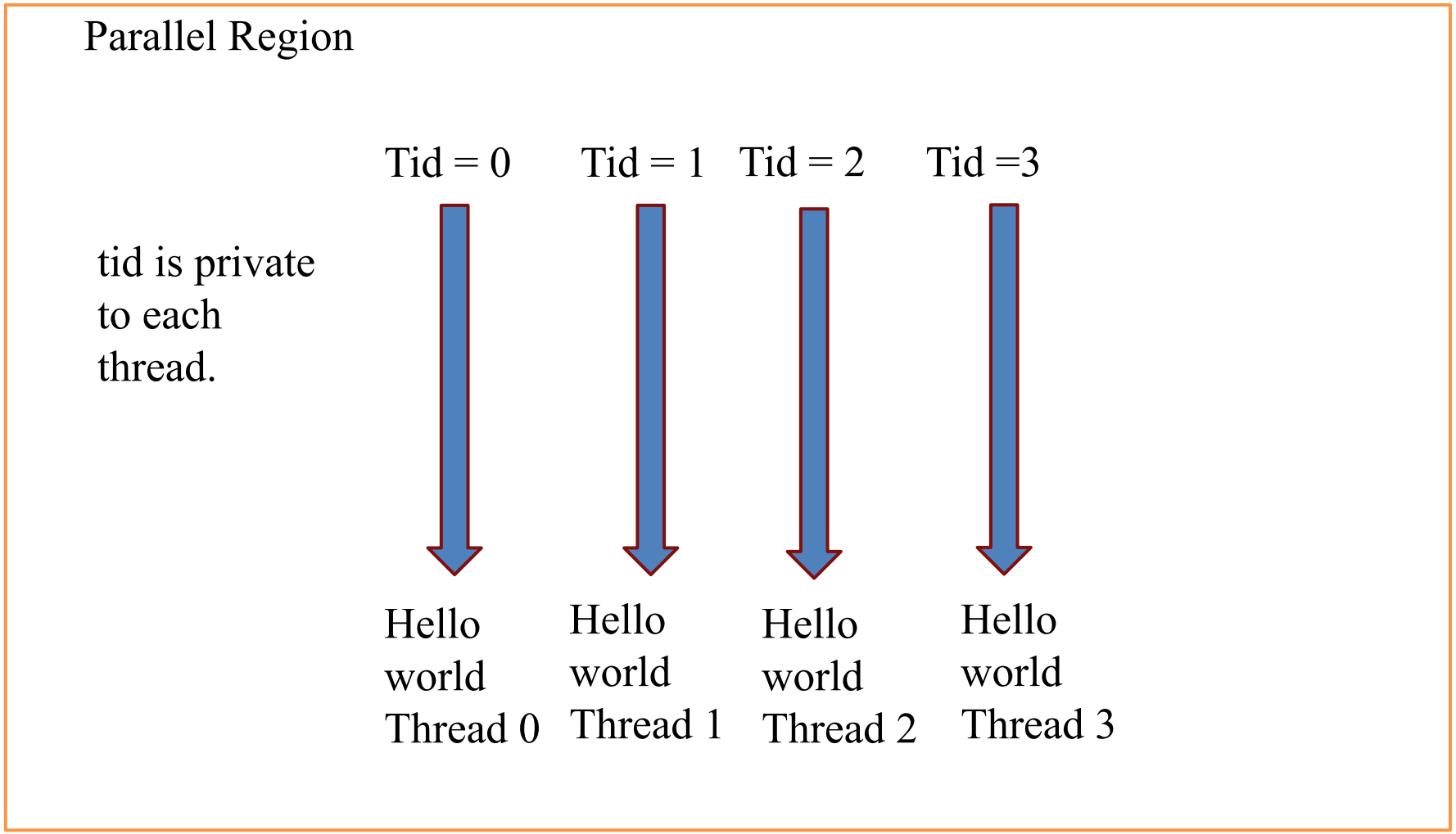
PGI: -mp

Parallel Software Models

Programing Models

-

Number of threads 4



Compile and execution

Compile program in OpenMP

- Set number of threads:

  Using shell

  $ export OMP_NUM_THREADS=4

  Inside program before parallel region.

  omp_set_num_threads(4);

- For GNU C compiler:

  $ gcc -fopenmp HelloWorld.c -o Hello

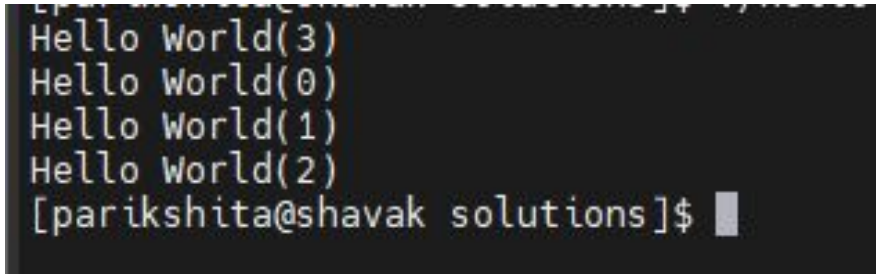
  $./Hello

- For Intel compiler:

  $ icc -qopenmp HelloWorld.c -o Hello

  $./Hello

Output of hello world program with 4 threads:



Thank you

OpenMP Components

Language extensions

OpenMP Components

- PARALLEL Region Construct.
  - Number of Threads
  - Nested Parallel Regions.
  - PARALLEL Region Example
  - Nested Parallel Region Example
- Work-sharing Constructs
  - Do/for
  - Single
  - Section
- Synchronization
- Run-time Library Routines
- Environment Variables

Directives

- Parallel region
- Worksharing constructs
- Tasking
- Offloading
- Affinity
- Error Handling
- SIMD
- Synchronization
- Data-Sharing attributes

Runtime Environment

- Number of threads
- Thread Id
- Dynamic thread adjustment
- Nested parallelism
- Schedule
- Thread limit
- Wallclock timer
- Team size
- Nesting level

Runtime Variables

- Number of threads
- Scheduling type
- Dynamic thread adjusting
- Nested parallelism
- Thread limit

Parallel Region Construct

- Block of code that will be executed by multiple threads
- Fundamental OpenMP parallel construct
- When a thread reaches a PARALLEL directive, it creates a team of threads and becomes the master of the team. The master is a member of that team and has thread number 0 within that team.
- Starting from the beginning of this parallel region, the code is duplicated and all threads will execute that code.
- There is an implied barrier at the end of a parallel section Only the master thread continues execution past this point.
- If any thread terminates within a parallel region, all threads in the team will terminate, and the work done up until that point is undefined.

Parallel Region Construct

#pragma omp parallel [clause[[,]clause] ...] new-line

clause:

if(scalar-expression)

num_threads(integer-expression)

default(shared | none)

private(list)

firstprivate(list)

shared(list)

reduction(operator: list)

Parallel Region Construct

Number of Threads

- The number of threads in a parallel region is determined by the following factors, in order of precedence:
  1. Evaluation of the IF clause
  2. Setting of the NUM_THREADS clause
  3. Use of the omp_set_num_threads() library function
  4. Setting of the OMP_NUM_THREADS environment variable
  5. Implementation default - usually the number of cores on a node.
- Threads are numbered from 0 (master thread) to N-1

PARALLEL Region Example

```
#include <omp.h>
#include <stdio.h>
int main (){
#pragma omp parallel
if(omp_in_parallel)
{
    printf("The threads is
%d\n",omp_get_thread_num());
}
return 0;
}
```

How to compile and run?
$gcc -fopenmp <file_name.c> -o <file_name>
$./<file_name>

```
[cdacapp2@login3 1.If]$ ./mp_if
The threads is 0
The threads is 2
The threads is 3
The threads is 1
```

Parallel Region Construct

Nested Parallel Regions

- Use the omp_get_nested() library function to determine if nested parallel regions are enabled.
- The two methods available for enabling nested parallel regions (if supported) are:
  - The omp_set_nested() library routine
  - Setting of the OMP_NESTED environment variable to TRUE
- If not supported, a parallel region nested within another parallel region results in the creation of a new team, consisting of one thread, by default.

PARALLEL Nested Region Example

```
#include <omp.h>
#include <stdio.h>
void report_num_threads(int level)
{
   #pragma omp single
   {
      printf("Level %d: number of threads in the team - %d\n",
             level, omp_get_num_threads());
   }
 }
int main()
{
   omp_set_dynamic(0);
   #pragma omp parallel num_threads(2)
   {
      report_num_threads(1);
      #pragma omp parallel num_threads(2)
      {
         report_num_threads(2);
         #pragma omp parallel num_threads(2)
         {
            report_num_threads(3);
         }
      }
   }
   return(0);
}
```

How to compile and run?

$gcc -fopenmp <file_name.c> -o <file_name>

$./<file_name>

```
(base) [cdacapp@login01 nested]$ export OMP_NESTED=True
(base) [cdacapp@login01 nested]$ ./nested
Level 1: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
```

Run-time Library Routines

- The OpenMP standard defines an API for library calls that perform a variety of functions:
  - Query the number of threads/processors, set number of threads to use
  - General purpose locking routines (semaphores)
  - Portable wall clock timing routines
  - Set execution environment functions: nested parallelism, dynamic adjustment of threads.
  - For C/C++, it may be necessary to specify the include file "omp.h".

Note: Your implementation may or may not support nested parallelism and/or dynamic threads. If nested parallelism is supported, it is often only nominal, in that a nested parallel region may only have one thread.

- To set number of threads during execution

  export OMP_NUM_THREDS=4

- To allow run time system to determine the number of threads

  export OMP_DYNAMIC=TRUE

- To all nesting of parallel region

  export OMP_NUSTED=TRUE

Data Scope Attribute Clauses

- Also called data sharing attribute clauses
- Because OpenMP is based upon the shared memory programming
- model, most variables are shared by default.
- Global variables include:
  - C: File scope variables, static
- Private variables include:
  - Loop index variables
- Clauses used to explicitly define how variables should be scoped
- include:
  - PRIVATE
  - FIRSTPRIVATE
  - LASTPRIVATE
  - SHARED
  - DEFAULT
  - REDUCTION
  - COPYIN

Data Scope Attribute Clauses (cont.)

- Used in conjunction with several directives (PARALLEL, DO/for, and SECTIONS) to control the scoping of enclosed variables.
- Provide the ability to control the data environment during execution of parallel constructs
  - Define how and which data variables in the serial section of the program are transferred to the parallel sections of the program (and back)
  - Define which variables will be visible to all threads in the parallel sections and which variables will be privately allocated to all threads
- Effective only within their lexical/static extent

PRIVATE and SHARED Clauses

- PRIVATE Clause
  - Declares variables in its list to be private to each thread
  - A new object of the same type is declared once for each thread in the team.
  - All references to the original object are replaced with references to the new object.
  - Variables declared PRIVATE should be assumed to be uninitialized for each thread.

- SHARED Clause
  - Declares variables in its list to be shared among all threads in the team
  - A shared variable exists in only one memory location and all threads can read or write to that address.
  - It is the programmer's responsibility to ensure that multiple threads properly access SHARED variables (such as via CRITICAL sections)

FIRSTPRIVATE and LASTPRIVATE Clauses

- FIRSTPRIVATE Clause
  - Combines the behaviour of the PRIVATE clause with automatic initialization of the variables in its list
  - Listed variables are initialized according to the value of their original objects prior to entry into the parallel or work-sharing construct.
- LASTPRIVATE Clause
  - Combines the behaviour of the PRIVATE clause with a copy from the last loop iteration or section to the original variable object
  - The value copied back into the original variable object is obtained from the last (sequentially) iteration or section of the enclosing construct.
  - For example, the team member that executes the final iteration

    for a DO section, or the team member that executes the last SECTION of a SECTIONS context, performs the copy with its own values.

DEFAULT Clause

- Allows the user to specify a default scope for all variables in the lexical extent of any parallel region
- Specific variables can be exempted from the default using the PRIVATE, SHARED, FIRSTPRIVATE, LASTPRIVATE, and REDUCTION clauses.
- The C/C++ OpenMP specification does not include private or firstprivate as a possible default. However, actual implementations may provide this option.
- Using NONE as a default requires that the programmer explicitly scope all variables.

Work-sharing Constructs

- A work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it.
- Work-sharing constructs do not launch new threads
- There is no implied barrier upon entry to a work-sharing construct, however there is an implied barrier at the end of a work sharing construct.
- A work-sharing construct must be enclosed dynamically within a parallel region in order for the directive to execute in parallel.
- Work-sharing constructs must be encountered by all members of a team or none at all
- Successive work-sharing constructs must be encountered in the same order by all members of a team.

Types of Work-sharing Constructs

- DO / for
  - Shares iterations of a loop across the team
  - Represents a type of "data parallelism”
- SECTIONS
  - Breaks work into separate, discrete sections
  - Each section is executed by a thread.
  - Can be used to implement a type of "functional parallelism”
- SINGLE
  - Serializes a section of code

For Directive Syntax

#pragma omp for [clause[[,] clause] ...] new-line

for-loops

clause:

private(list)

firstprivate(list)

lastprivate(list)

reduction(operator: list)

schedule(kind[, chunk_size])

ordered

nowait

For loop

The for construct tells OpenMP that the iteration set of the for loop that follows is to be distributed across the threads present in the team. Without the for construct, the entire iteration set of the for loop concerned will be executed by each thread in the team.

The for directive supports the following clauses:

- **private**
- **firstprivate**
- **lastprivate**
- **reduction**
- **ordered**
- **schedule**
- **Nowait**

- How iteration are divide in threads?

0 - N-1

For loop - PRIVATE

- The values of private data are undefined upon entry to and exit from the specific construct.
- Loop iteration variable is private by default.

How to compile and run?
$gcc -fopenmp <file_name.c> -o <file_name>
$./<file_name>

```
$ ./for_private
The thread 0 value is 0
The thread 0 value is 30
The thread 1 value is 0
The thread 1 value is 30
The thread 3 value is 0
The thread 3 value is 30
The thread 4 value is 0
The thread 4 value is 30
The thread 2 value is 0
The thread 2 value is 30
```

For loop - PRIVATE

```c
#include<stdio.h>
#include<omp.h>
#define N  5
int main()
{
    int a = 10; //shared
    int b = 20; //shared
    int c = 20;
    omp_set_num_threads(N);
    #pragma omp parallel for private(c)
    for(int i = 0; i < N;i++){
        printf("The thred %d value is %d\n",omp_get_thread_num(),c);
        c = a + b;    //private
        printf("The thred %d value is %d\n",omp_get_thread_num(),c);
    }
    return 0;
}
```

For loop - FIRSTPRIVATE

- The clause combines behaviour of private clause with automatic initialization of the variables in its list.

How to compile and run?
$gcc -fopenmp <file_name.c> -o <file_name>
$./<file_name>

```
$ ./for_firstprivate
The thred 0 value is 20
The thred 0 value is 30
The thred 4 value is 20
The thred 4 value is 30
The thred 3 value is 20
The thred 3 value is 30
The thred 2 value is 20
The thred 2 value is 30
The thred 1 value is 20
The thred 1 value is 30
```

For loop - FIRSTPRIVATE

```c
#include<stdio.h>
#include<omp.h>
#define N  5
int main()
{
    int a = 10; //shared
    int b = 20; //shared
    int c = 20;
    omp_set_num_threads(N);
    #pragma omp parallel for firstprivate(c)
    for(int i = 0; i < N;i++){
        printf("The thred %d value is %d\n",omp_get_thread_num(),c);
        c = a + b;    //private
        printf("The thred %d value is %d\n",omp_get_thread_num(),c);
    }
    return 0;
}
```

For loop - LASTPRIVATE

- Performs finalization of private variables
- Each thread has its own copy

How to compile and run?
$gcc -fopenmp <file_name.c> -o <file_name>
$./<file_name>

```
$ ./for_lastprivate
The thred 4 value is 30
The thred 2 value is 30
The thred 1 value is 30
The thred 3 value is 30
The thred 0 value is 30
The thred 0 value is 30
```

For loop - LASTPRIVATE

```c
#include<stdio.h>
#include<omp.h>
#define N  5
int main()
{
    int a = 10; //shared
    int b = 20; //shared
    int c ;
    omp_set_num_threads(N);
    #pragma omp parallel for lastprivate(c)
    for(int i = 0; i < N;i++){
        c = a + b;    //private
        printf("The thred %d value is %d\n",omp_get_thread_num(),c);
    }
    printf("The thred %d value is %d\n",omp_get_thread_num(),c);
    return 0;
}
```

REDUCTION Clause

- Performs a reduction on the variables that appear in its list.
- A private copy for each list variable is created for each thread. At the end of the reduction, the reduction operator is applied to all private copies of the shared variable, and the final result is written to the global shared variable.
- Syntax:

  reduction (operator: list)

For loop - REDUCTION

- The reduction clause indicates that the variables passed are, as its name suggests, used in a reduction.
- By default, the reduction computation is complete at the end of the construct.

How to compile and run?
$gcc -fopenmp <file_name.c> -o <file_name>
$./<file_name>

$ ./reduction
Number of primes numbers: 5761455

For loop - REDUCTION

```
#include <stdio.h>
#include <omp.h>
#define NUM 100000000
int isprime( int x )
{
      for( int y = 2; y * y <= x; y++ )
            {
                  if( x % y == 0 )
                        return 0;
                  }
          return 1;
}
int main( )
{
      int sum = 0;
#pragma omp parallel for reduction (+:sum)
          for( int i = 2; i <= NUM ; i++ )
                {
                      sum += isprime ( i );
                      }
              printf( "Number of primes numbers: %d\n", sum );
                  return 0;
}
```

For loop - ORDERED

- The omp ordered directive identifies a structured block of code that must be executed in sequential order.

How to compile and run?
$gcc -fopenmp <file_name.c> -o <file_name>
$./<file_name>

```
$ ./ordered
Thread 1 processes iteration 3.
Thread 1 processes iteration 4.
Thread 1 processes iteration 5.
Thread 2 processes iteration 6.
Thread 2 processes iteration 7.
Thread 0 processes iteration 0.
Thread 0 processes iteration 1.
Thread 0 processes iteration 2.
Thread 3 processes iteration 8.
Thread 3 processes iteration 9.
```

For loop - ORDERED

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

int main(int argc, char* argv[])
{
    omp_set_num_threads(4);
    #pragma omp parallel for ordered
    for(int i = 0; i < 10; i++)
    {
        printf("Thread %d processes iteration %d.\n", omp_get_thread_num(), i);
    }

    return 0;
}
```

For loop - SCHEDULE

- The schedule clause tells OpenMP how to distribute the loop iterations to the threads.

  The OpenMP scheduling kind to use.
  Possible values:

  - auto: the auto scheduling kind will apply.
  - dynamic: the dynamic scheduling kind will apply.
  - guided: the guided scheduling kind will apply.
  - runtime: the runtime scheduling kind will apply.
  - static: the static scheduling kind will apply.

SCHEDULE Clause.. cont

schedule(static, [n])

- Each thread is assigned chunks in “round robin” fashion, known as block cyclic scheduling.
- divides iterations into chunks that are approximately equal in size and it distributes at most one chunk to each thread
- If n has not been specified, it will contain CEILING(number_of_iterations / number_of_threads) iterations

  Example: loop of length 16, 3 threads, chunk size 2 :

For loop - SCHEDULE(static)

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

int main(int argc, char* argv[])
{
    omp_set_num_threads(3);

    printf("With no chunksize passed:\n");
    #pragma omp parallel for schedule(static)
    for(int i = 0; i < 10; i++)
    {
        printf("Thread %d processes iteration %d.\n", omp_get_thread_num(), i);
    }
    printf("With a chunksize of 2:\n");

    #pragma omp parallel for schedule(static, 2)
    for(int i = 0; i < 10; i++)
    {
        printf("Thread %d processes iteration %d.\n", omp_get_thread_num(), i);
    }

    return 0;
}
```

```
$ ./parallel_for
With no chunksize passed:
Thread 0 processes iteration 0.
Thread 0 processes iteration 1.
Thread 0 processes iteration 2.
Thread 2 processes iteration 7.
Thread 2 processes iteration 8.
Thread 2 processes iteration 9.
Thread 0 processes iteration 3.
Thread 1 processes iteration 4.
Thread 1 processes iteration 5.
Thread 1 processes iteration 6.
```

```
With a chunksize of 2:
Thread 1 processes iteration 2.
Thread 1 processes iteration 3.
Thread 1 processes iteration 8.
Thread 1 processes iteration 9.
Thread 2 processes iteration 4.
Thread 2 processes iteration 5.
Thread 0 processes iteration 0.
Thread 0 processes iteration 1.
Thread 0 processes iteration 6.
Thread 0 processes iteration 7.
```

How to compile and run?
$gcc -fopenmp <file_name.c> -o <file_name>
$./<file_name>

Static

The nice thing with static scheduling is that OpenMP run-time guarantees that if you have two separate loops with the same number of iterations and execute them with the same number of threads using static scheduling, then each thread will receive exactly the same iteration range(s) in both parallel regions.

This is quite important on NUMA systems: if you touch some memory in the first loop, it will reside on the NUMA node where the executing thread was. Then in the second loop the same thread could access the same memory location faster since it will reside on the same NUMA node.

For loop - SCHEDULE(dynamic)

schedule(dynamic, [n])

- Divides the iterations into chunks of equal size, but each thread can be assigned a different chunk of iterations dynamically at runtime until no chunk remain to be distributed.

- The chunk size is determined by the chunksize parameter or by the compiler.

- There is no particular order in which the chunks are distributed to the threads. The order changes each time when we execute the for loop.

For loop - SCHEDULE(dynamic

schedule(dynamic, [n])

- Iteration of loop are divided into chunks containing n iterations each.

```
$ ./parallel_for_loop
With no chunksize passed:
Thread 0 processes iteration 0.
Thread 0 processes iteration 2.
Thread 0 processes iteration 3.
Thread 0 processes iteration 4.
Thread 0 processes iteration 5.
Thread 0 processes iteration 6.
Thread 0 processes iteration 7.
Thread 0 processes iteration 8.
Thread 0 processes iteration 9.
Thread 1 processes iteration 1.
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(int argc, char* argv[])
{
    omp_set_num_threads(2);
    printf("With no chunksize passed:\n");
    #pragma omp parallel for schedule(dynamic)
    for(int i = 0; i < 10; i++)
    {
        printf("Thread %d processes iteration %d.\n", omp_get_threa
    }
    printf("With a chunksize of 2:\n");
    #pragma omp parallel for schedule(dynamic, 2)
    for(int i = 0; i < 10; i++)
    {
        printf("Thread %d processes iteration %d.\n", omp_get_threa
    }
    return 0;
}
```

How to compile and run?
$gcc -fopenmp <file_name.c> -o <file_name>
$./<file_name>

```
With a chunksize of 2:
Thread 1 processes iteration 0.
Thread 1 processes iteration 1.
Thread 1 processes iteration 4.
Thread 1 processes iteration 5.
Thread 1 processes iteration 6.
Thread 1 processes iteration 7.
Thread 1 processes iteration 8.
Thread 1 processes iteration 9.
Thread 0 processes iteration 2.
Thread 0 processes iteration 3.
```

- Default chunk size is 1.

SCHEDULE Clause.. cont

schedule(guided, [n])

- If you specify n, that is the minimum chunk size that each thread should posses..
- Size of each successive chunk is exponentially decreasing.
- The size of a chunk is proportional to the number of unassigned iterations divided by the number of the threads. Therefore the size of the chunks decreases.
- Initial chunk size
  max(number_of_iterations / number_of_threads,n)
  Subsequent chunk consist of
  max(remaining_iterations/number_of_threds,n) iterations

  Schedule(runtime):export OMP_SCHEDULE “STATIC,4”
- Determine the scheduling type at run time by the OMP_SCHEDULE environment variable.

For loop - SCHEDULE(guided)

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

int main(int argc, char* argv[])
{
    omp_set_num_threads(4);
    #pragma omp parallel for schedule(guided)
    for(int i = 0; i < 10; i++)
    {
        printf("Thread %d processes iteration %d.\n", omp_get_thread_num(), i);
    }

    return 0;
}
```

```
$ ./parallel_loop_scheduling
Thread 2 processes iteration 7.
Thread 2 processes iteration 8.
Thread 2 processes iteration 9.
Thread 1 processes iteration 5.
Thread 1 processes iteration 6.
Thread 3 processes iteration 3.
Thread 3 processes iteration 4.
Thread 0 processes iteration 0.
Thread 0 processes iteration 1.
Thread 0 processes iteration 2.
```

How to compile and run?
$gcc -fopenmp <file_name.c> -o <file_name>
$./<file_name>

For loop - SCHEDULE(runtime)

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

int main(int argc, char* argv[])
{
    omp_set_num_threads(4);
    #pragma omp parallel for schedule(runtime)
    for(int i = 0; i < 10; i++)
    {
        printf("Thread %d processes iteration %d.\n", omp_get_thread_num(), i);
    }

    return 0;
}
```

```
$./parallel_for_runtime
Thread 0 processes iteration 0.
Thread 0 processes iteration 4.
Thread 0 processes iteration 5.
Thread 0 processes iteration 6.
Thread 0 processes iteration 7.
Thread 0 processes iteration 8.
Thread 1 processes iteration 3.
Thread 2 processes iteration 2.
Thread 0 processes iteration 9.
Thread 3 processes iteration 1.
```

How to compile and run?
$gcc -fopenmp <file_name.c> -o <file_name>
$./<file_name>

SECTIONS Construct

- The sections construct is a non-iterative worksharing construct that contains a set of structured blocks.
- The block are distributed among and executed by the threads in a team.
- Each structured block is executed once by one of the threads in the team in the context of its implicit task.

The parallel directive supports the following clauses:

- **private(list)**
- **firstprivate(list)**
- **lastprivate([modifier :] list)**
- **reduction([modifier,] identifier : list)**
- **nowait**

SECTIONS - PRIVATE

```
$./sections
section 2 # Working thread : 2 | 0.000000 * 22.350000 = 0.000000
section 2 # Working thread : 2 | 1.500000 * 23.350000 = 35.025002
section 2 # Working thread : 2 | 3.000000 * 24.350000 = 73.050003
section 2 # Working thread : 2 | 4.500000 * 25.350000 = 114.075005
section 2 # Working thread : 2 | 6.000000 * 26.350000 = 158.100006
section 1 # Working thread : 0 | 0.000000 + 22.350000 = 22.350000
section 1 # Working thread : 0 | 1.500000 + 23.350000 = 24.850000
section 1 # Working thread : 0 | 3.000000 + 24.350000 = 27.350000
section 1 # Working thread : 0 | 4.500000 + 25.350000 = 29.850000
section 1 # Working thread : 0 | 6.000000 + 26.350000 = 32.349998
```

```c
#include<stdio.h>
#include <omp.h>
#define N     5
int main () {
int i;
float a[N], b[N], c[N], d[N],e[N];
/* Some initializations */
for (i=0; i < N; i++) {
  a[i] = i * 1.5;
  b[i] = i + 22.35;
  }
/* Fork a team of threads with each thread having a private i variable and
#pragma omp parallel shared(a,b,c,d,e) private(i)
  {
  #pragma omp sections
    {
    #pragma omp section
    for (i=0; i < N; i++){
      c[i] = a[i] + b[i];
      printf("section 1 # Working thread : %d | %f + %f = %f\n",omp_get_thread_num(),a[i],b[i],c[i]);
    }
    #pragma omp section
    for (i=0; i < N; i++){
      d[i] = a[i] * b[i];
      printf("section 2 # Working thread : %d | %f * %f = %f\n",omp_get_thread_num(),a[i],b[i],d[i]);
    }
    }  /* end of sections */
  }  /* end of parallel section */
 return 0;
}
```

How to compile and run?

$gcc -fopenmp <file_name.c> -o <file_name>

$./<file_name>

SECTIONS - FIRSTPRIVATE

```
$ ./sections
section 2 # Working thread : 2 | 0.000000 * 22.350000 = 0.000000
section 2 # Working thread : 2 | 1.500000 * 23.350000 = 35.025002
section 2 # Working thread : 2 | 3.000000 * 24.350000 = 73.050003
section 2 # Working thread : 2 | 4.500000 * 25.350000 = 114.075005
section 2 # Working thread : 2 | 6.000000 * 26.350000 = 158.100006
section 1 # Working thread : 0 | 0.000000 + 22.350000 = 22.350000
section 1 # Working thread : 0 | 1.500000 + 23.350000 = 24.850000
section 1 # Working thread : 0 | 3.000000 + 24.350000 = 27.350000
section 1 # Working thread : 0 | 4.500000 + 25.350000 = 29.850000
section 1 # Working thread : 0 | 6.000000 + 26.350000 = 32.349998
```

```c
#include<stdio.h>
#include <omp.h>
#define N     5
int main () {
int i;
float a[N], b[N], c[N], d[N],e[N];
/* Some initializations */
for (i=0; i < N; i++) {
  a[i] = i * 1.5;
  b[i] = i + 22.35;
  }
/* Fork a team of threads with each thread having a private i variable and shared
#pragma omp parallel shared(a,b,c,d,e) firstprivate(i)
  {
  #pragma omp sections nowait
    {
    #pragma omp section
    for (i=0; i < N; i++){
      c[i] = a[i] + b[i];
      printf("section 1 # Working thread : %d | %f + %f = %f\n",omp_get_thread_num(),a[i],b[i],c[i]);
    }
    #pragma omp section
    for (i=0; i < N; i++){
      d[i] = a[i] * b[i];
      printf("section 2 # Working thread : %d | %f * %f = %f\n",omp_get_thread_num(),a[i],b
    }
    }  /* end of sections */
  }  /* end of parallel section */
 return 0;
}
```

How to compile and run?

$gcc -fopenmp <file_name.c> -o <file_name>

$./<file_name>

SECTIONS - LASTPRIVA

```c
#include<stdio.h>
#include <omp.h>
#define N     5
int main () {
int i;
float a[N], b[N], c[N], d[N],e[N];
/* Some initializations */
for (i=0; i < N; i++) {
  a[i] = i * 1.5;
  b[i] = i + 22.35;
  }
/* Fork a team of threads with each thread having a private i varia
#pragma omp parallel shared(a,b,c,d,e)
  {
  #pragma omp sections nowait lastprivate(i)
    {
    #pragma omp section
    for (i=0; i < N; i++){
      c[i] = a[i] + b[i];
      printf("section 1 # Working thread : %d | %f + %f = %f\n",omp_get_thread_num(),a[i],b[i],c[i]);
    }
    #pragma omp section
    for (i=0; i < N; i++){
      d[i] = a[i] * b[i];
      printf("section 2 # Working thread : %d | %f * %f = %f\n",omp_get_thread_num(),a[i],b[i],d[i]);
    }
    }  /* end of sections */
  }  /* end of parallel section */
 return 0;
```

```
$ ./sections
section 1 # Working thread : 0 | 0.000000 + 22.350000 = 22.350000
section 1 # Working thread : 0 | 1.500000 + 23.350000 = 24.850000
section 1 # Working thread : 0 | 3.000000 + 24.350000 = 27.350000
section 1 # Working thread : 0 | 4.500000 + 25.350000 = 29.850000
section 1 # Working thread : 0 | 6.000000 + 26.350000 = 32.349998
section 2 # Working thread : 1 | 0.000000 * 22.350000 = 0.000000
section 2 # Working thread : 1 | 1.500000 * 23.350000 = 35.025002
section 2 # Working thread : 1 | 3.000000 * 24.350000 = 73.050003
section 2 # Working thread : 1 | 4.500000 * 25.350000 = 114.075005
section 2 # Working thread : 1 | 6.000000 * 26.350000 = 158.100006
```

How to compile and run?
$gcc -fopenmp <file_name.c> -o <file_name>
$./<file_name>

SECTIONS - REDUCTION

```c
#include <stdio.h>
#include <omp.h>
#define NUM 100
int isprime( int x )
{
      for( int y = 2; y * y <= x; y++ )
            {
                  if( x % y == 0 )
                        return 0;
                        }
          return 1;
}
int main( )
{
      int sum = 0;
      int i;
#pragma omp parallel
  {
  #pragma omp sections reduction (+:sum)
    {
    #pragma omp section
      {
          for( int i = 2; i <= NUM ; i++ )
                {
                      sum += isprime ( i );
                        }
              printf( "Number of primes numbers: %d\n", sum );
      }
   #pragma omp section
      {
          for( int i = 2; i <= NUM ; i++ )
                {
                      sum += isprime ( i );
                        }
              printf( "Number of primes numbers: %d\n", sum );
      }
    }

 }
       return 0;
}
```

How to compile and run?
$gcc -fopenmp <file_name.c> -o <file_name>
$./<file_name>

```
$ ./reduction
Number of primes numbers: 25
Number of primes numbers: 25
```

SECTIONS - NOWAIT

- In OpenMP, the nowait clause is used to indicate that a thread should not wait for other threads at the end of a parallel region or a parallel loop construct. By default, when a parallel region or a parallel loop completes, the threads synchronize and wait for each other before continuing execution.

- The nowait clause allows threads to continue their execution immediately after completing their portion of work without waiting for other threads to finish. This can improve performance in situations where subsequent code does not depend on the completion of all threads.

SECTIONS - NOWAIT

```
$ ./sections
section 2 # Working thread : 1 | 0.000000 * 22.350000 = 0.000000
section 2 # Working thread : 1 | 1.500000 * 23.350000 = 35.025002
section 2 # Working thread : 1 | 3.000000 * 24.350000 = 73.050003
section 2 # Working thread : 1 | 4.500000 * 25.350000 = 114.075005
section 2 # Working thread : 1 | 6.000000 * 26.350000 = 158.100006
section 1 # Working thread : 0 | 0.000000 + 22.350000 = 22.350000
section 1 # Working thread : 0 | 1.500000 + 23.350000 = 24.850000
section 1 # Working thread : 0 | 3.000000 + 24.350000 = 27.350000
section 1 # Working thread : 0 | 4.500000 + 25.350000 = 29.850000
section 1 # Working thread : 0 | 6.000000 + 26.350000 = 32.349998
```

```c
#include<stdio.h>
#include <omp.h>
#define N     5
int main () {
int i;
float a[N], b[N], c[N], d[N],e[N];
/* Some initializations */
for (i=0; i < N; i++) {
  a[i] = i * 1.5;
  b[i] = i + 22.35;
  }
/* Fork a team of threads with each thread having a private i var
#pragma omp parallel shared(a,b,c,d,e) private(i)
  {
  #pragma omp sections nowait
    {
    #pragma omp section
    for (i=0; i < N; i++){
      c[i] = a[i] + b[i];
      printf("section 1 # Working thread : %d | %f + %f = %f\n",omp_get_thread_num(),a[i],b[i],c[i]);
    }
    #pragma omp section
    for (i=0; i < N; i++){
      d[i] = a[i] * b[i];
      printf("section 2 # Working thread : %d | %f * %f = %f\n",omp_get_thread_num(),a[i],b[i],d[i]);
    }
    }  /* end of sections */
  }  /* end of parallel section */
 return 0;
}

```

How to compile and run?
$gcc -fopenmp <file_name.c> -o <file_name>
$./<file_name>

SINGLE Construct

- single is a clause that must be used in a parallel region; it tells OpenMP that the associated block must be executed by one thread only, .

  The clause to appear on the single construct, which is one of the following:
  - **private(list)**
  - **firstprivate(list)**
  - **copyprivate(list)**
  - **nowait**

- master will be executed by the master only while single can be executed by whichever thread reaching the region first.

SINGLE-PRIVATE

```
$ ./single
1st loop # Working thread : 0 # 0.000000 + 22.350000 = 22.350000
1st loop # Working thread : 0 # 1.500000 + 23.350000 = 24.850000
1st loop # Working thread : 0 # 3.000000 + 24.350000 = 27.350000
1st loop # Working thread : 0 # 4.500000 + 25.350000 = 29.850000
1st loop # Working thread : 0 # 6.000000 + 26.350000 = 32.349998
2nd loop # Working thread : 0 # 0.000000 * 22.350000 = 0.000000
2nd loop # Working thread : 0 # 1.500000 * 23.350000 = 35.025002
2nd loop # Working thread : 0 # 3.000000 * 24.350000 = 73.050003
2nd loop # Working thread : 0 # 4.500000 * 25.350000 = 114.075005
2nd loop # Working thread : 0 # 6.000000 * 26.350000 = 158.100006
```

```c
#include<stdio.h>
#include <omp.h>
#define N     5
int main () {
int i;
float a[N], b[N], c[N], d[N];
/* Some initializations */
for (i=0; i < N; i++) {
  a[i] = i * 1.5;
  b[i] = i + 22.35;
  }
  /* Fork a team of threads with each thread having a private i variable and shared varable a,b,c,d */
#pragma omp parallel shared(a,b,c,d) private(i)
  {
  #pragma omp single

    {
    for (i=0; i < N; i++){
      c[i] = a[i] + b[i];
      printf("1st loop # Working thread : %d # %f + %f = %f\n",omp_get_thread_num(),a[i],b[i],c[i]);
     }
    for (i=0; i < N; i++){
      d[i] = a[i] * b[i];
      printf("2nd loop # Working thread : %d # %f * %f = %f\n",omp_get_thread_num(),a[i],b[i],d[i]);
    }
    }
    /* end of sections */
  }  /* end of parallel section */
 return 0;
}
```

How to compile and run?
$gcc -fopenmp <file_name.c> -o <file_name>
$./<file_name>

SINGLE-FIRSTPRIVATE

```c
#include<stdio.h>
#include <omp.h>
#define N     5
int main () {
int i = 1;
float a[N], b[N], c[N], d[N];
/* Some initializations */
for (i=0; i < N; i++) {
  a[i] = i * 1.5;
  b[i] = i + 22.35;
  }
  /* Fork a team of threads with each thread having a private i v
#pragma omp parallel shared(a,b,c,d) firstprivate(i)
  {
  #pragma omp single

    {
    for (i=0; i < N; i++){
      c[i] = a[i] + b[i];
      printf("1st loop # Working thread : %d # %f + %f = %f\n",omp_get_thread_num(),a[i],b[i],c[i]);
     }
    for (i=0; i < N; i++){
      d[i] = a[i] * b[i];
      printf("2nd loop # Working thread : %d # %f * %f = %f\n",omp_get_thread_num(),a[i],b[i],d[i]);
    }
    }
    /* end of sections */
  }  /* end of parallel section */
 return 0;
}
```

```
$ ./single
1st loop # Working thread : 0 # 0.000000 + 22.350000 = 22.350000
1st loop # Working thread : 0 # 1.500000 + 23.350000 = 24.850000
1st loop # Working thread : 0 # 3.000000 + 24.350000 = 27.350000
1st loop # Working thread : 0 # 4.500000 + 25.350000 = 29.850000
1st loop # Working thread : 0 # 6.000000 + 26.350000 = 32.349998
2nd loop # Working thread : 0 # 0.000000 * 22.350000 = 0.000000
2nd loop # Working thread : 0 # 1.500000 * 23.350000 = 35.025002
2nd loop # Working thread : 0 # 3.000000 * 24.350000 = 73.050003
2nd loop # Working thread : 0 # 4.500000 * 25.350000 = 114.075005
2nd loop # Working thread : 0 # 6.000000 * 26.350000 = 158.100006
```

How to compile and run?
$gcc -fopenmp <file_name.c> -o <file_name>
$./<file_name>

SINGLE-COPYPRIVATE

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(int argc, char* argv[])
{
    int a = 123;

    #pragma omp parallel default(none) firstprivate(a)
    {
        printf("Thread %d: a = %d.\n", omp_get_thread_num(), a);

        #pragma omp barrier

        #pragma omp single copyprivate(a)
        {
            a = 456;
            printf("Thread %d executes the single construct and changes a to %d.\n", omp_get_thread_num(), a);
        }

        printf("Thread %d: a = %d.\n", omp_get_thread_num(), a);
    }

    return 0;
}
```

```
$ ./mp_copyprivate
Thread 0: a = 123.
Thread 1: a = 123.
Thread 2: a = 123.
Thread 3: a = 123.
Thread 0 executes the single construct and
changes a to 456.
Thread 2: a = 456.
Thread 3: a = 456.
Thread 1: a = 456.
Thread 0: a = 456.
```

How to compile and run?
$gcc -fopenmp <file_name.c> -o <file_name>
$./<file_name>

SINGLE-NOWAIT

```c
#include<stdio.h>
#include <omp.h>
#define N     5
int main () {
int i;
float a[N], b[N], c[N], d[N];
/* Some initializations */
for (i=0; i < N; i++) {
  a[i] = i * 1.5;
  b[i] = i + 22.35;
  }
  /* Fork a team of threads with each thread having a private i variable and shared varable a,b,c,d */
#pragma omp parallel shared(a,b,c,d) private(i)
  {
  #pragma omp single nowait

    {
    for (i=0; i < N; i++){
      c[i] = a[i] + b[i];
      printf("1st loop # Working thread : %d # %f + %f = %f\n",omp_get_thread_num(),a[i],b[i],c[i]);
     }
    for (i=0; i < N; i++){
      d[i] = a[i] * b[i];
      printf("2nd loop # Working thread : %d # %f * %f = %f\n",omp_get_thread_num(),a[i],b[i],d[i]);
    }
    }
    /* end of sections */
  }  /* end of parallel section */
 return 0;
}
```

How to compile and run?
$gcc -fopenmp <file_name.c> -o <file_name>
$./<file_name>

```
$ ./single
1st loop # Working thread : 0 # 0.000000 + 22.350000
= 22.350000
1st loop # Working thread : 0 # 1.500000 + 23.350000
= 24.850000
1st loop # Working thread : 0 # 3.000000 + 24.350000
= 27.350000
1st loop # Working thread : 0 # 4.500000 + 25.350000
= 29.850000
1st loop # Working thread : 0 # 6.000000 + 26.350000
= 32.349998
2nd loop # Working thread : 0 # 0.000000 * 22.350000
= 0.000000
2nd loop # Working thread : 0 # 1.500000 * 23.350000
= 35.025002
2nd loop # Working thread : 0 # 3.000000 * 24.350000
= 73.050003
2nd loop # Working thread : 0 # 4.500000 * 25.350000
= 114.075005
2nd loop # Working thread : 0 # 6.000000 * 26.350000
= 158.100006
```

Questions for Thought

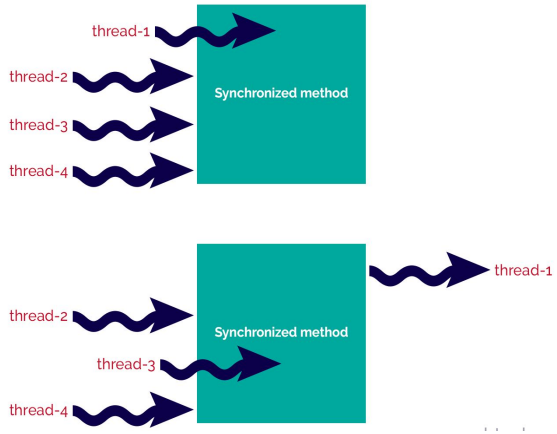
- What happens if the number of threads and the number of SECTIONs are different? What if there are more threads than SECTIONs? fewer threads than SECTIONs?
- Which thread executes which SECTION?

Synchronization Constructs

- Motivation: Consider a simple example where two threads on two different processors are both trying to increment a variable x at the same time (assume x is initially 0).

```
THREAD 1:
increment(x)
{
 x = x + 1;
}
THREAD 1:
10 LOAD A, (x address)
20 ADD A, 1
30 STORE A, (x address)
```

```
THREAD 2:
increment(x)
{
 x = x + 1;
}
THREAD 2:
10 LOAD A, (x address)
 20 ADD A, 1
 30 STORE A, (x address)
```



- The incrementation of x must be synchronized between the two threads to insure that the correct result is produced.
- OpenMP provides a variety of synchronization constructs that control how the execution of each thread proceeds relative to other team threads.

Critical Construct

The critical construct, which is used inside parallel regions, tells OpenMP that the associated block is to be executed by every thread but no more than one thread at a time.

How to compile and run?
$gcc -fopenmp <file_name.c> -o <file_name>
$./<file_name>

```
$./critical
[Thread 0] Total before I add my value (1): 0.
[Thread 0] Total after I added my value: 1.
[Thread 3] Total before I add my value (6): 1.
[Thread 3] Total after I added my value: 7.
[Thread 2] Total before I add my value (2): 7.
[Thread 2] Total after I added my value: 9.
[Thread 1] Total before I add my value (1): 9.
[Thread 1] Total after I added my value: 10.
```

Critical Construct

```c
int main(int argc, char* argv[])
{
    // Use 4 threads when creating OpenMP parallel regions
    omp_set_num_threads(4);

    int total = 0;

    // Create the parallel region
    #pragma omp parallel default(none) shared(total)
    {
        // Calculate my factorial
        int my_value = 1;
        for(int i = 2; i <= omp_get_thread_num(); i++)
        {
            my_value *= i;
        }

        // Add my value to the total
        #pragma omp critical
        {
            printf("[Thread %d] Total before I add my value (%d): %d.\n", omp_get_thread_num(), my_value, total);
            total += my_value;
            printf("[Thread %d] Total after I added my value: %d.\n", omp_get_thread_num(), total);
        }
    }

    return 0;
}
```

Master Construct

master is a clause that must be used in a parallel region; it tells OpenMP that the associated block must be executed by the master thread only. The other threads do not wait at the end of the associated block as if there was an implicit barrier.

How to compile and run?
$gcc -fopenmp <file_name.c> -o <file_name>
$./<file_name>

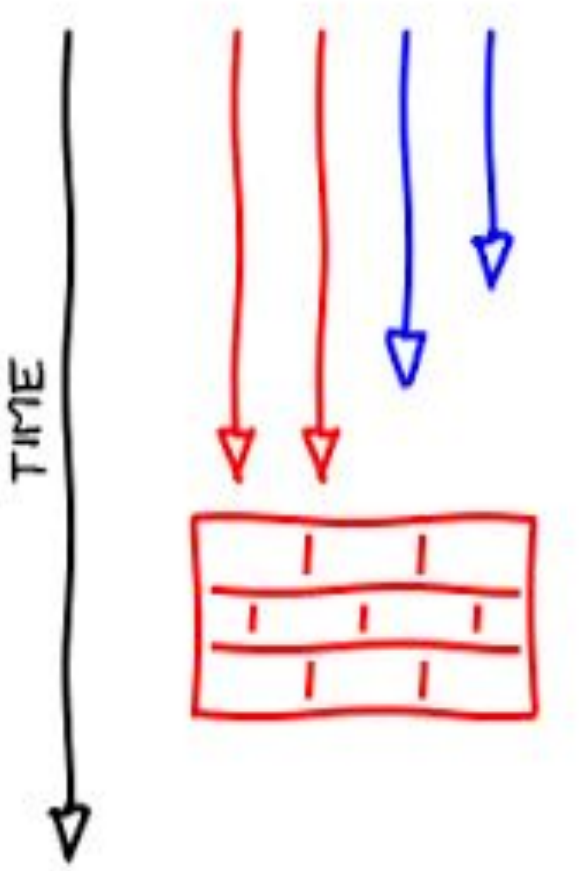
$ ./Master
[Thread 0] Every thread executes this printf.
[Thread 3] Every thread executes this printf.
[Thread 1] Every thread executes this printf.
[Thread 2] Every thread executes this printf.
[Thread 0] Only the master thread executes this printf, which is me.

Master Construct

```c
int main(int argc, char* argv[])
{
    // Use 4 threads when creating OpenMP parallel regions
    omp_set_num_threads(4);

    // Create the parallel region
    #pragma omp parallel
    {
        printf("[Thread %d] Every thread executes this printf.\n", omp_get_thread_num());

        #pragma omp barrier

        #pragma omp master
        {
            printf("[Thread %d] Only the master thread executes this printf, which is me.\n", omp_get_thread_num());
        }
    }

    return 0;
}
```

BARRIER Construct

- Explicit wait for other threads to Complete their task



How to compile and run?
$gcc -fopenmp <file_name.c> -o <file_name>
$./<file_name>

```
$ ./barrier
 [Thread 0] I print my first message.
[Thread 3] I print my first message.
[Thread 1] I print my first message.
[Thread 2] I print my first message.
The barrier is complete, which means all threads have printed their first message.
[Thread 3] I print my second message.
[Thread 2] I print my second message.
[Thread 1] I print my second message.
[Thread 0] I print my second message.
```

BARRIER

```c
int main(int argc, char* argv[])
{
    // Use 4 threads when we create a parallel region
    omp_set_num_threads(4);

    // Create the parallel region
    #pragma omp parallel
    {
        // Threads print their first message
        printf("[Thread %d] I print my first message.\n", omp_get_thread_num());

        // Make sure all threads have printed their first message before moving on.
        #pragma omp barrier

        // One thread indicates that the barrier is complete.
        #pragma omp single
        {
            printf("The barrier is complete, which means all threads have printed their first message.\n");
        }

        // Threads print their second message
        printf("[Thread %d] I print my second message.\n", omp_get_thread_num());
    }

    return 0;
}

```

ATOMIC

In OpenMP, the atomic construct is used to ensure that a specific memory location or variable is accessed atomically, without interference from other threads. It guarantees that the read-modify-write operation on the variable is performed as a single, indivisible unit, preventing race conditions.

The atomic construct can be applied to a wide range of operations, such as read, write, update, or arithmetic operations. It provides a simple and efficient way to enforce atomicity without requiring explicit locks or critical sections.

ATOMIC

- The atomic construct ensures that a specific storage location is accessed atomically, rather than exposing it to the possibility of multiple, simultaneous reading and writing threads that may result in indeterminate values.

```c
int main(int argc, char* argv[])
{
    // Use 4 threads when creating OpenMP parallel
    omp_set_num_threads(4);
    int total = 0;
    // Create the parallel region
    #pragma omp parallel default(none) shared(total)
    {
        for(int i = 0; i < 10; i++)
        {
            // Atomically add one to the total
            #pragma omp atomic
            total++;
        }
    }
    printf("Total = %d.\n", total);
    return 0;
}
```

How to compile and run?
$gcc -fopenmp <file_name.c> -o <file_name>
$./<file_name>

```
$ ./atomic
Total = 40.
```

FLUSH Directive

- Identifies a synchronization point at which the implementation must provide a consistent view of memory
- Thread-visible variables are written back to memory at this point.
- Necessary to instruct the compiler that a variable must be written to/read from the memory system, i.e. that a variable cannot be kept in a local CPU register
  - Keeping a variable in a register in a loop is very common when producing efficient machine language code for a loop

C/C++: #pragma omp flush (list) newline

FLUSH Directive (cont.)

- The optional list contains a list of named variables that will be flushed in order to avoid flushing all variables. For pointers in the list, the pointer itself is flushed, not the object to which it points.
- Implementations must ensure any prior modifications to thread-visible variables are visible to all threads after this point; i.e., compilers must restore values from registers to memory, hardware might need to flush write buffers, etc.
- The FLUSH directive is implied for the directives shown in the table below. The directive is not implied if a NOWAIT clause is present

FLUSH Directive (cont.)

- The FLUSH directive is implied for the directives shown in the table below. The directive is not implied if a NOWAIT clause is present.

C/C++
barrier
parallel – upon entry and exit
critical – upon entry and exit
ordered – upon entry and exit
for – upon exit
sections – upon exit
single – upon exit

ORDERED Directive

- Specifies that iterations of the enclosed loop will be executed in the same order as if they were executed on a serial processor
- Threads will need to wait before executing their chunk of iterations if previous iterations haven't completed yet.
- Used within a DO / for loop with an ORDERED clause
- The ORDERED directive provides a way to "fine tune" where ordering is to be applied within a loop. Otherwise, it is not required.
- An ORDERED directive can only appear in the dynamic extent of the following directives:
  - DO or PARALLEL DO (Fortran)
  - for or parallel for (C/C++)
- Only one thread is allowed in an ordered section at any time
- An iteration of a loop must not execute the same ORDERED directive more than once, and it must not execute more than one ORDERED directive.
- A loop that contains an ORDERED directive must be a loop with an ORDERED clause.

ORDERED Example

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

int main(void)
{
#pragma omp parallel for ordered
for (int i = 1; i <= 8; i++)
{
    #pragma omp ordered
    printf("Thread %d is executing iteration %d \n", omp_get_thread_num(), i);
}
    return 0;
}
```

How to compile and run?
$gcc -fopenmp <file_name.c> -o <file_name>
$./<file_name>

```
(base) [cdacapp@login03 10.Ordered]$ ./ordered
Thread 0 is executing iteration 1
Thread 1 is executing iteration 2
Thread 2 is executing iteration 3
Thread 3 is executing iteration 4
Thread 4 is executing iteration 5
Thread 5 is executing iteration 6
Thread 6 is executing iteration 7
Thread 7 is executing iteration 8
(base) [cdacapp@login03 10.Ordered]$
```

DEFAULT Clause

- Allows the user to specify a default scope for all variables in the lexical extent of any parallel region
- Specific variables can be exempted from the default using the PRIVATE, SHARED, FIRSTPRIVATE, LASTPRIVATE, and REDUCTION clauses.
- The C/C++ OpenMP specification does not include private or firstprivate as a possible default. However, actual implementations may provide this option.
- Using NONE as a default requires that the programmer explicitly scope all variables.

THREADPRIVATE Directive

- Used to make global file scope variables (C/C++) or common blocks (Fortran) local and persistent to a thread through the execution of multiple parallel regions
- Must appear after the declaration of listed variables/common blocks
- Each thread then gets its own copy of the variable/common block, so data written by one thread is not visible to other threads.
- On first entry to a parallel region, data in THREADPRIVATE variables and common blocks should be assumed undefined, unless a COPYIN clause is specified in the PARALLEL directive
- THREADPRIVATE variables differ from PRIVATE variables because they are able to persist between different parallel sections of a code.
- Data in THREADPRIVATE objects is guaranteed to persist only if the dynamic threads mechanism is "turned off" and the number of threads in different parallel regions remains constant. The default setting of dynamic threads is undefined.

THREADPRIVATE Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int a = 12345;
#pragma omp threadprivate(a)
int main(int argc, char* argv[])
{
    // Turn off dynamic threads as required by threadprivate
    omp_set_dynamic(0);
    #pragma omp parallel copyin(a)
    {
        #pragma omp master
        {
            printf("[First parallel region] Master thread changes the value of a to 67890.\n");
            a = 67890;
        }
        #pragma omp barrier
        printf("[First parallel region] Thread %d: a = %d.\n", omp_get_thread_num(), a);
    }
    #pragma omp parallel copyin(a)
    {
        printf("[Second parallel region] Thread %d: a = %d.\n", omp_get_thread_num(), a);
    }
    return 0;
}
```

```
(base) [cdacapp@login03 11.Threadprvate]$ ./copyin
[First parallel region] Master thread changes the value of a to 67890.
[First parallel region] Thread 4: a = 12345.
[First parallel region] Thread 1: a = 12345.
[First parallel region] Thread 3: a = 12345.
[First parallel region] Thread 2: a = 12345.
[First parallel region] Thread 0: a = 67890.
[Second parallel region] Thread 1: a = 67890.
[Second parallel region] Thread 0: a = 67890.
[Second parallel region] Thread 2: a = 67890.
[Second parallel region] Thread 3: a = 67890.
[Second parallel region] Thread 4: a = 67890.
(base) [cdacapp@login03 11.Threadprvate]$
```

How to compile and run?
$gcc -fopenmp <file_name.c> -o <file_name>
$./<file_name>

COPYIN Clause

- Provides a means for assigning the same value to THREADPRIVATE variables for all threads in the team
- List contains the names of variables to copy. In Fortran, the list can contain both the names of common blocks and named variables.
- The master thread variable is used as the copy source. The team threads are initialized with its value upon entry into the parallel construct

Run-time Library Routines

- The OpenMP standard defines an API for library calls that perform a variety of functions:
  - Query the number of threads/processors, set number of threads to use
  - General purpose locking routines (semaphores)
  - Portable wall clock timing routines
  - Set execution environment functions: nested parallelism, dynamic adjustment of threads.
  - For C/C++, it may be necessary to specify the include file "omp.h".

Note: Your implementation may or may not support nested parallelism and/or dynamic threads. If nested parallelism is supported, it is often only nominal, in that a nested parallel region may only have one thread.

- To set number of threads during execution

  export OMP_NUM_THREDS=4

- To allow run time system to determine the number of threads

  export OMP_DYNAMIC=TRUE

- To all nesting of parallel region

  export OMP_NUSTED=TRUE

Thank You

OpenMP Tasking

Remember OpenMP Sections?

Sections are independent blocks of code, able to be assigned to separate threads if they are available.

```
#pragma omp parallel sections
{
    #pragma omp section
    {
        Task 1
    }
    #pragma omp section
    {
        Task 2
    }
}
```

There is an implied barrier at the end

OpenMP sections are static, that is, they are good if you know, when you are writing the program, how many of them you will need.

It would be nice to have something more Dynamic

Imagine a capability where you can write something to do down on a Post-It®note, accumulate the Post-It notes, then have all of the threads together execute that set of tasks.

You would also like to not have to know, ahead of time, how many of these Post-It notes you will write. That is, you want the total number to be dynamic. Well,

congratulations, you have just invented OpenMP Tasks!

Terminology

- **Task :** A specific instance of executable code and its data environment that the OpenMP implementation can schedule for execution by threads.
- **Task region:** A region consisting of all code encountered during the execution of a task.
- **Implicit task:** A task generated by an implicit parallel region or generated when a parallel construct is encountered during execution.
- **Initial task:** An implicit task associated with an implicit parallel region.
- **Explicit task:** A task that is not an implicit task.

Tasking

OpenMP 3.0 introduced a new feature called tasking.

Tasking allows for the parallelization of applications where units of work are generated dynamically, such as in recursive structures or while loops, without having to rely on nested parallelism. This simplifies the logic for the programmer as well as reduces the overhead inherent in creating multiple, nested, parallel regions

Tasking

Rather than organizing all tasks into their own groups like in sections, tasking allows for more unstructured parallelism, as when a task pragma is encountered by a thread in a parallel region, it is placed into a task queue and can be executed by a thread as soon as one becomes available.

The scheduling of tasks, which each contain the code to execute, the task data environment, and internal control variables, to threads, is handled by the OpenMP runtime system .

The initial creation of tasks is often handled by a single thread as will be shown in the discussion of the sorting implementations to follow. Synchronization between tasks is achieved using the taskwait pragma. This construct specifies a wait on the completion of child tasks generated since the beginning of the current task and allows for synchronization between dependent tasks.

Tasking execution model

Supports unstructured parallelism

- unbounded loops

```
while ( <expr> ) {
...
}
```

- recursive functions

```
void myfunc( <args> )
{
...; myfunc( <newargs> ); ...;
}
```

- Several scenarios are possible:
  - single creator, usually by means of parallel / single
  - multiple creators, work-sharing and nested tasks

- All threads in the team are candidates to execute tasks

Task construct

```
#pragma omp task [clause[ [,] clause] ... ] new-line
    structured-block
```

- may execute the task **immediately** or
- **defer its execution** to one of the other threads in the team.
- A task is always **bound to the innermost parallel region**.
- If a task construct is encountered outside a parallel region, then the structured block is executed immediately by the encountering thread.

A serial implementation in C looks like:

```
 void mergesort_serial(int a[], int n, int temp[]) {
    mergesort_serial(a, n/2, temp);
    mergesort_serial(a + n/2, n - n/2, temp);
    merge(a, n, temp);
}
```

A parallel implementation in OpenMP using sections looks like this:

```
void mergesort_parallel
      (int a[], int n, int temp[], int thrds){
    if ( threads == 1) {
        mergesort_serial(a, n, temp);
    }
    else if (threads > 1) {
    #pragma omp parallel sections
        {
     #pragma omp section
      mergesort_parallel(a, n/2,temp,thrds/2);
     #pragma omp section
      mergesort_parallel(a + n/2, n - n/2,
                temp + n/2, thrds - thrds/2);
        }
    merge(a, size, temp);
    }
}
```

A parallel mergesort using OpenMP tasking looks like:

```
void mergesort_parallel(int a[], int n, temp[], int thrds, int thresh) {
  if (threads == 1) {
    mergesort_serial(a, n, temp, thresh);
  }
  else if (threads > 1) {
    #pragma omp parallel
    {
      #pragma omp single nowait
      {
        #pragma omp task
        {
          mergesort_par(a, n / 2, temp, thrds/2, thresh);
        }
        #pragma omp task
        {
          mergesort_parallel_omp(a + size / 2, n - n / 2,
          temp + n/2, thrds – thrds /     2,thresh);
        }
```

```
if (p < r) {
    int q = partition (p, r, data);
    #pragma omp task firstprivate(data, low_limit, r, q)
      quick_sort (p, q - 1, data, low_limit);
    #pragma omp task firstprivate(data, low_limit, r, q)
     quick_sort (q + 1, r, data, low_limit);
  }
}

void par_quick_sort (int n, int *data, int low_limit) {
  #pragma omp parallel
  {
    #pragma omp single nowait
      quick_sort (0, n, data, low_limit);
  }
}
```

Note the two separate functions. The main function will call par_quick_sort() which creates a single parallel region and has a single thread call quick_sort() which then generates the initial tasks. These tasks then generate child tasks recursively until the list is sorted. See how nested parallelism is not a requirement to achieve full thread utilization. A single parallel region is created with all available system threads and tasks will be assigned to the threads as they are created. Since the sort is done in place, each task or section needs its own copy of the data which is why the firstprivate clause is used.

Task construct(Cont)

The task construct accepts a set of clauses:

if([task :] scalar-expression)
final(scalar-expression)
untied
default(shared | none)
mergeable
private(list)
firstprivate(list)
shared(list)
in_reduction(reduction-identifier : list)
depend([depend-modifier,] dependence-type : locator-list)
priority(priority-value)
allocate([allocator :] list)
affinity([aff-modifier :] locator-list)
detach(event-handle)

Task scheduling: tied vs untied tasks

- Tasks are tied by default (when no untied clause present)
  - tied tasks are executed always by the same thread (not necessarily creator)
  - tied tasks may run into performance problems

- **#pragma omp task untied**
  {structured-block}
  - bad mix with thread based features: thread-id, threadprivate, critical regions...
  - gives the runtime more flexibility to schedule tasks
  - but most of OpenMP implementations doesn't “honor” untied

Tied Tasks:

- Tied tasks are the default behavior in OpenMP.
- In tied task scheduling, each thread that creates a task is responsible for executing it.
- Once a thread creates a task, it is bound to that task until the task completes.
- Tied tasks guarantee that a thread executes the task it creates, providing locality of reference.

Untied Tasks:

- Untied tasks allow more flexibility in task execution.
- In untied task scheduling, the thread that creates a task may not necessarily be the one that executes it.
- The OpenMP runtime system dynamically assigns tasks to available threads, promoting load balancing and potentially improving performance.
- Threads that finish executing their tasks can steal tasks from other threads' task queues, reducing potential idle time.

Choosing Between Tied and Untied Tasks:

Tied tasks are suitable when the task creator thread possesses relevant data and context needed to execute the task effectively. This can provide better locality and reduce communication overhead.

Untied tasks are beneficial when load balancing is crucial or when the workload of tasks is unpredictable. The runtime system can distribute tasks among available threads, potentially reducing idle time.

Hello World Time Again

Note that the task pragma is inside a parallel construct. Each thread in the team.

- encounters the task construct,
- creates the corresponding task and
- either executes the task immediately or defer its execution to one of the other threads in the team:

Therefore, the number of tasks, and lines printed, are the same as the number of threads in the team:

```
#include <stdio.h>

int main() {

    #pragma omp parallel

    {

        #pragma omp task

        printf("Hello world!\n");
    }
    return 0;
}
```

Data sharing rules

How to compile and run?
$gcc -fopenmp <file_name.c> -o <file_name>
$./<file_name>

```
(base) [cdacapp@login04 Task]$ ./data_sharing
1.I think the number is 1.
0.I think the number is 1.
3.I think the number is 1.
2.I think the number is 1.
```

```
#include <stdio.h>
int main() {
    #pragma omp parallel
    {
        int number = 1;
        #pragma omp task
        {
            printf("I think the number is %d.\n",
number);
            number++;
        }
    }
    return 0;
}
```

Challenge 1

Q. Modify the following program such that it uses explicit data sharing rules and the incrementation (number++) is protected with a critical construct:

Hint: You may want to write the value of the number variable to a private variable.

```
#include <stdio.h>

int main() {
    int number = 1;
    #pragma omp parallel
    {
        #pragma omp task
        {
            printf("I think the number is %d.\n",
number);
            number++;
        }
    }
    return 0;
}
```

Challenge 2

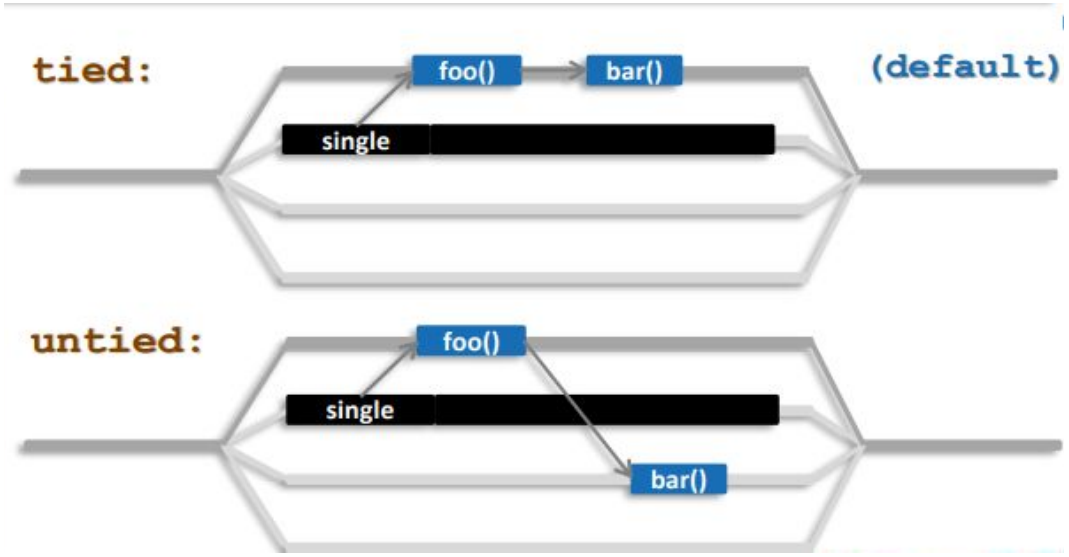
Write a program that creates 10 tasks. Each task should print the thread number of the calling thread.

- Task scheduling points (and the taskyield directive)
  - tasks can be suspended/resumed at TSPs some additional constraints to avoid deadlock problems
  - implicit scheduling points (creation, synchronization, ...)
  - explicit scheduling point: the taskyield directive

#pragma omp taskyield

- Scheduling [tied/untied] tasks: example

```
#pragma omp parallel
#pragma omp single
{
      #pragma omp task
            {
                  foo();
                  #pragma omp taskyield
                  bar()
            }
}
```



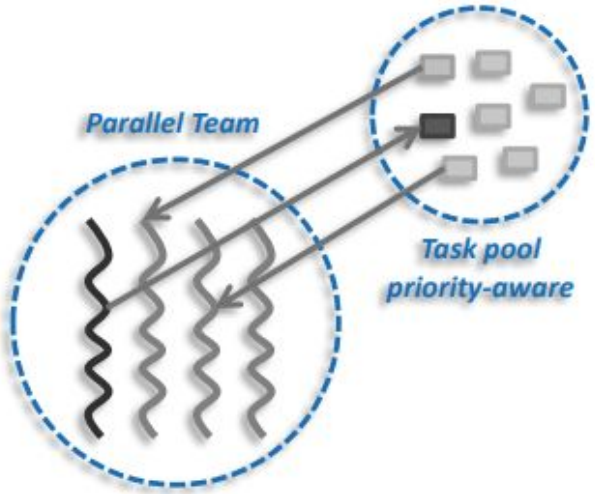
Task scheduling: programmer's hints

- Programmers may specify a priority value when creating a task

```
#pragma omp task priority(pvalue)
      {structured-block}
```

  - pvalue: the higher the best (will be scheduled earlier)
  - once a thread becomes idle, gets one of the highest priority tasks

```
#pragma omp parallel
#pragma omp single
{
for ( i = 0; i < SIZE; i++) {
#pragma omp task priority(1)
{ code_A; }
}
#pragma omp task priority(100)
{ code_B; }
...
}
```



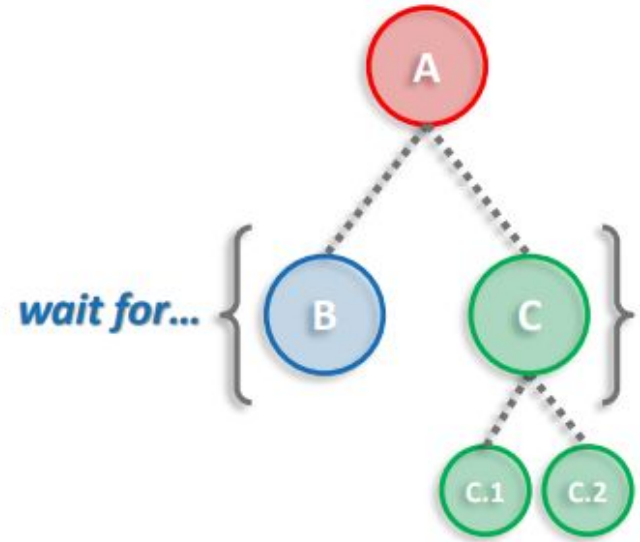
Task synchronization: taskwait directive

- The taskwait directive (shallow task synchronization)
  - It is a stand-alone directive

```
#pragma omp taskwait
```

  - wait on the completion of child tasks of the current task; just direct children, not all descendant tasks; includes an implicit task scheduling point (TSP)

```
#pragma omp parallel
#pragma omp single
{
        #pragma omp task                    :A
        {
                #pragma omp task    :B
                { … }
                #pragma omp task    :C
                { … …}
        #pragma omp taskwait
        }
} // implicit barrier will wait for C.x
```



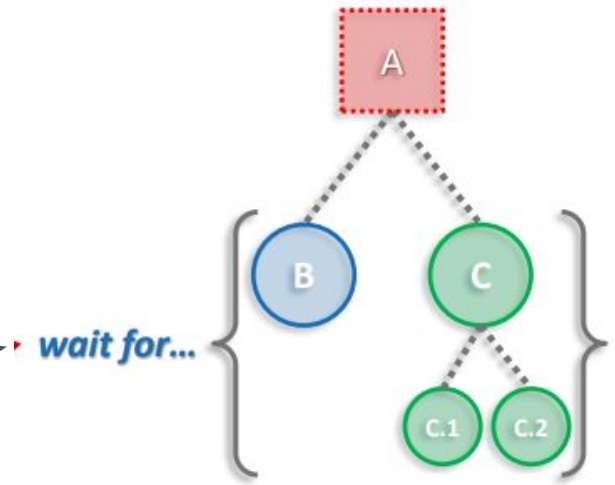
Task synchronization: taskgroup construct

- The taskgroup construct (deep task synchronization)
  - attached to a structured block; completion of all descendants of the current task; TSP at the end

```
#pragma omp taskgroup [clause[[,] clause]...]
{structured-block}
```

  - where clause (could only be): reduction(reduction-identifier: list-items)

```
#pragma omp parallel
#pragma omp single
{
#pragma omp taskgroup
        {
        #pragma omp task
        { … }
        #pragma omp task
        { … #C.1; #C.2; …}
} // end of taskgroup
}
```



Explicit data-sharing clauses

- Explicit data-sharing clauses (shared, private and firstprivate)

```
#pragma omp task
shared(a)
{
// Scope of a: shared
}
```

```
#pragma omp task private(b)
{
// Scope of b: private
}
```

```
#pragma omp task firstprivate(c)
{
// Scope of c: firstprivate
}
```

  - If default clause present, what the clause says
    - shared: data which is not explicitly included in any other data sharing clause will be shared
    - none: compiler will issue an error if the attribute is not explicitly set by the programmer (very useful!!!)

```
#pragma omp task default(shared)
{
// Scope of all the references, not explicitly
// included in any other data sharing clause,
// and with no pre-determined attribute: shared
}
```

```
#pragma omp task default(none)
{ // Compiler will force to specify the scope for
// every single variable referenced in the
context }
Hint: Use default(none) to be forced to think
about every variable if you do not see clearly
```

Implicit data-sharing attributes (in-practice)

- Implicit data-sharing attributes (in-practice)
  - the shared attribute is lexically inherited
  - in any other case the variable is firstprivate

```
int a = 1;
void foo() {
int b = 2, c = 3;
#pragma omp parallel private(b)
{
int d = 4;
#pragma omp task
        {
                int e = 5;
                // Scope of a: shared
                // Scope of b: firstprivate
                // Scope of c: shared
                // Scope of d: firstprivate
                // Scope of e: private
        }
        }
}
```

- Pre-determined rules (can not change)
- Explicit data-sharing clauses (+ default)
- Implicit data-sharing rules

- (in-practice) variable values within the task:
  - value of a: 1
  - value of b: x // undefined (undefined in parallel)
  - value of c: 3
  - value of d: 4
  - value of e: 5

Demo

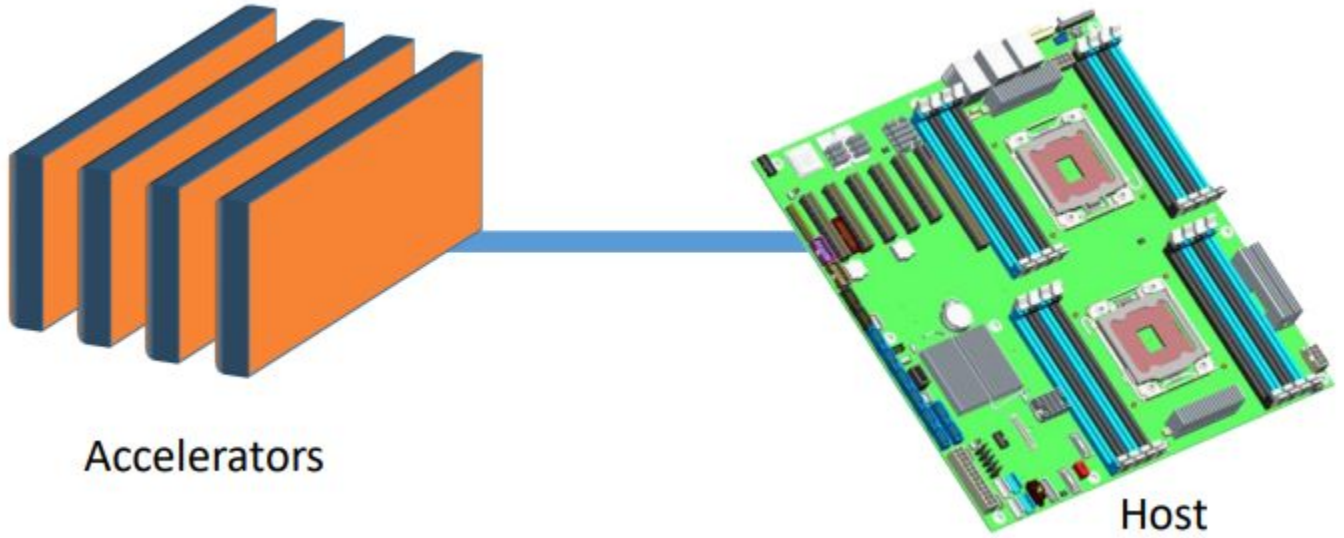
Thank You

OPENMP OFFLOADING

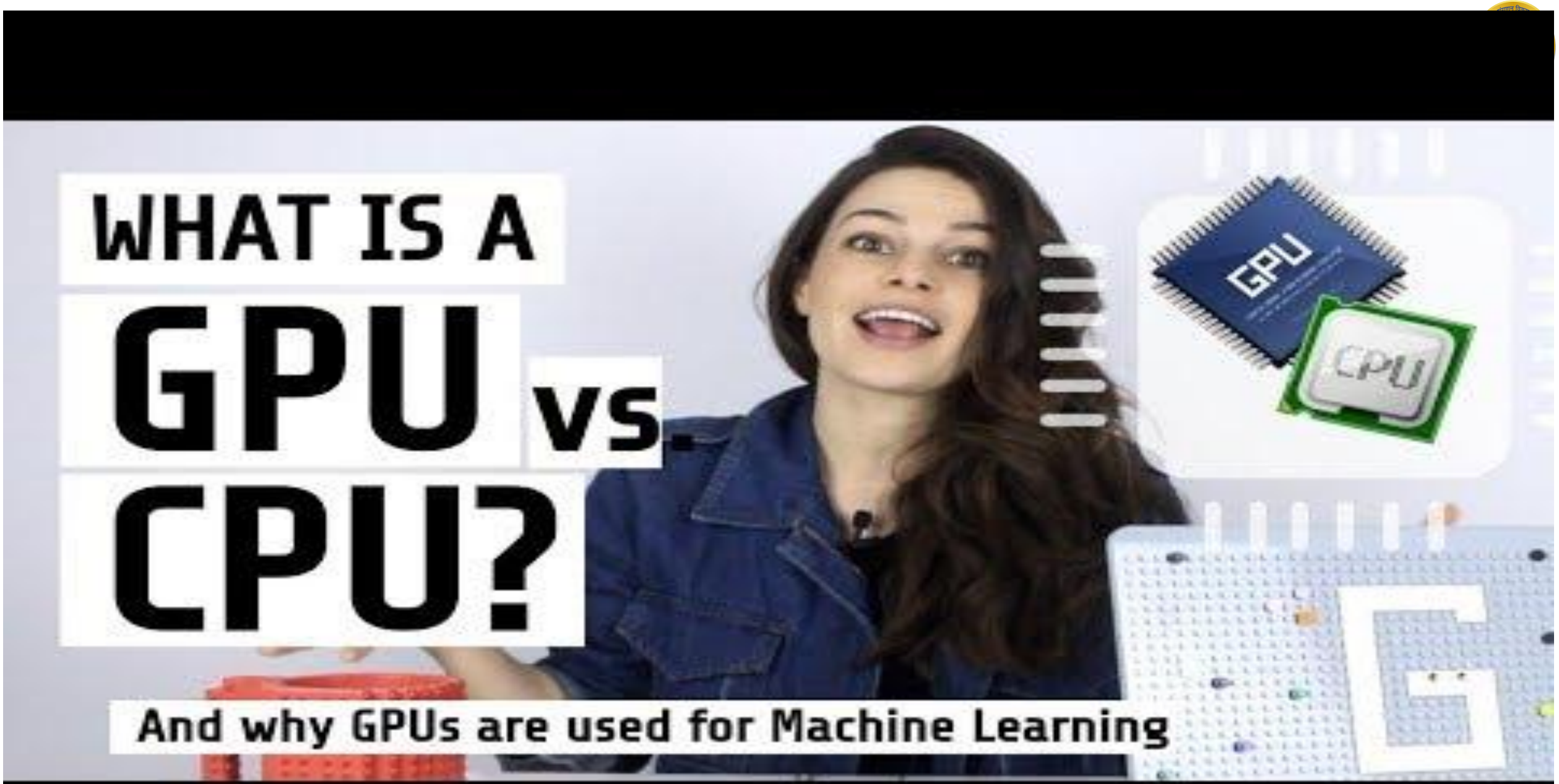
GPU Work

What is offloading ?

- Offloading is the process of transferring a portion of a computation from a central processor (such as a CPU) to a specialized accelerator (such as a coprocessor , FPGA or GPU) or to a remote machine.
- It can be use to improve the performance of a computation

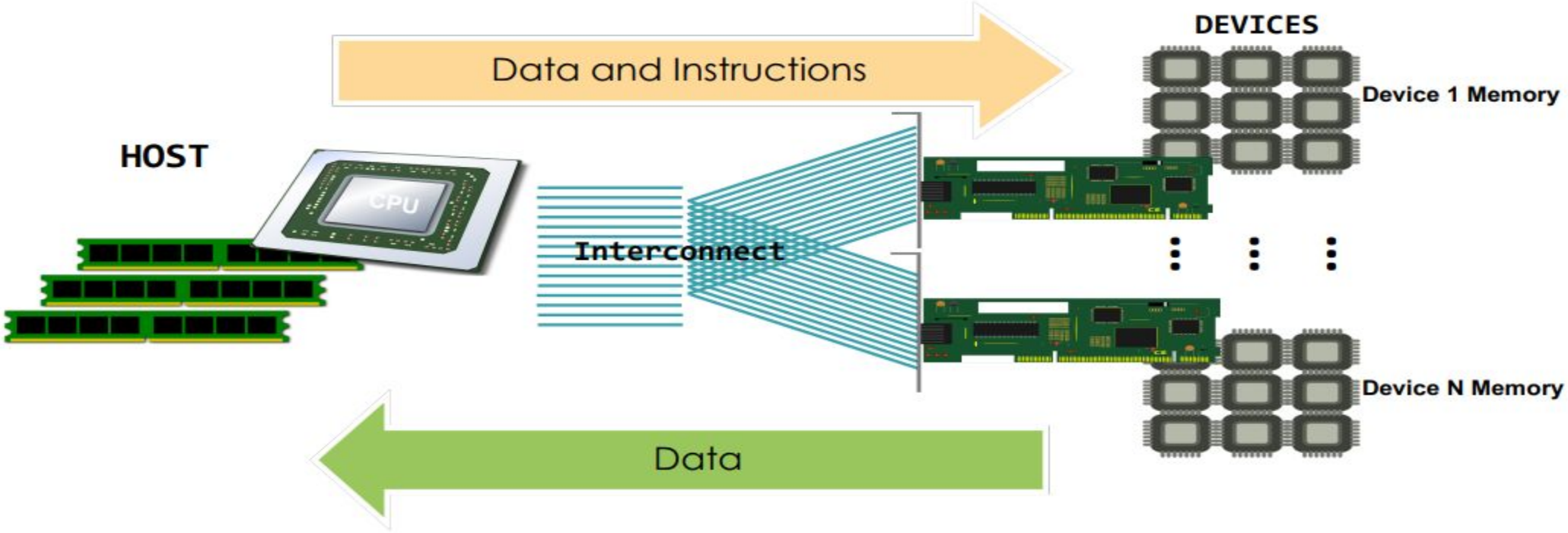


What is the Difference Between GPU and CPU?


WHAT IS A
GPU vs.
CPU?
And why GPUs are used for Machine Learning
GPU
CPU

Introduction: OpenMP Offload

OpenMP offload constructs are a set of directives for C++ and Fortran that were introduced in OpenMP 4.0 and further enhanced in later versions.



OpenMP Offload: Steps

- Identification of compute kernels
  - CPU initiates kernel for execution on the device

- Expressing parallelism within the kernel

- Manage data transfer between CPU and Device
  - relevant data needs to be moved from host to device memory
  - kernel executes using device memory
  - relevant data needs to be moved from device to main memory

Step 1: Identification of Kernels to Offload

- Look for compute intensive code and that can benefit from parallel Execution
  - Use performance analysis tools to find bottlenecks

- Track independent work units with well defined data accesses

- Keep an eye on platform specs
  - GPU memory is a precious resource

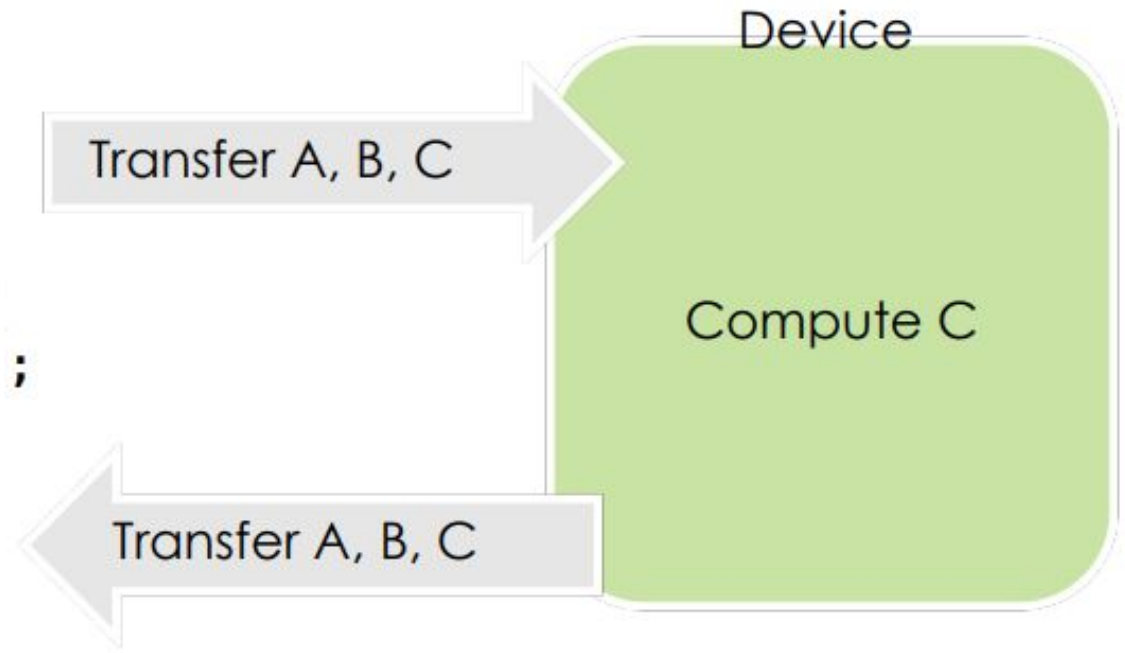
How to Offload ?

C/C++	Description
#pragma omp target [clause[[,] clause] ...] newline structured-block	The target construct offloads the enclosed code to the accelerator.

– A device data environment is created for the structured block
  – The code region is mapped to the device and executed.

Example using omp target

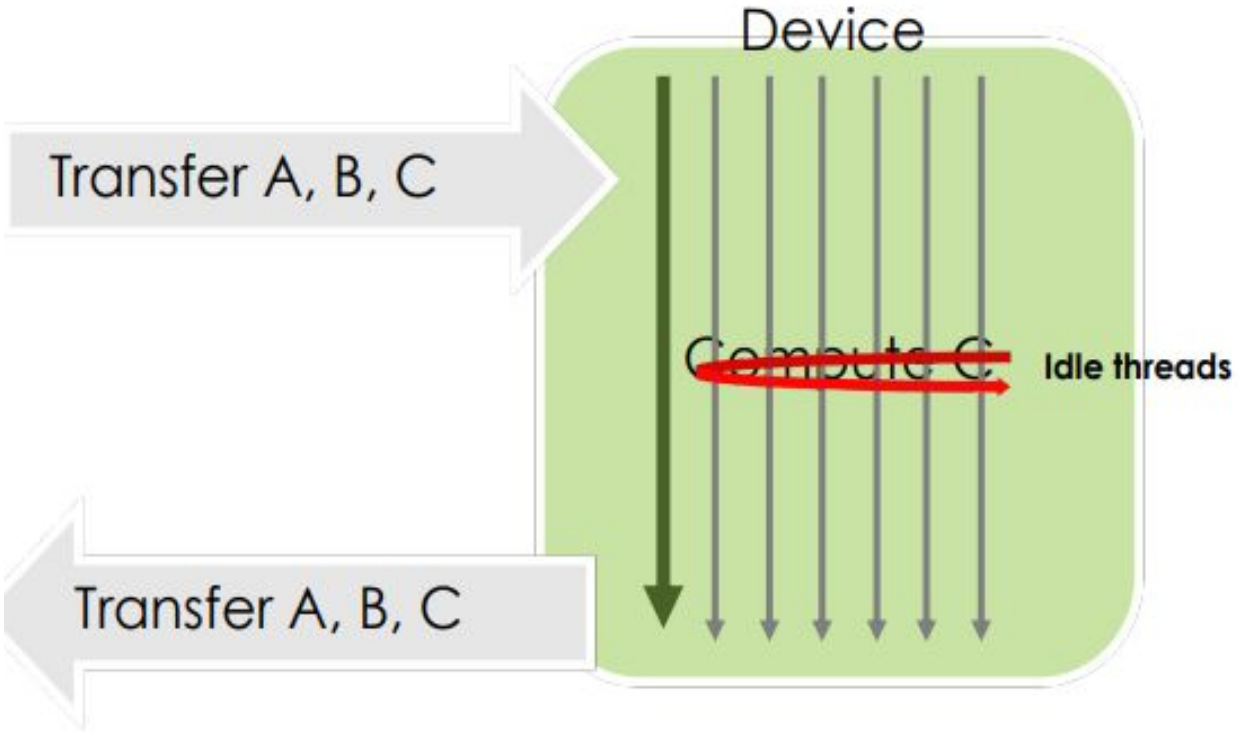
```
int A[N][N], B[N][N], C[N][N];
/* initialize arrays */
#pragma omp target
{
      for (int i = 0; i < N; ++i) {
      for (int j = 0; j < N; ++j) {
      C[i][j] = A[i][j] + B[i][j];
}
}
} // end target
```



The target construct is a task generating construct

Example using omp target(Cont.)

```
int A[N][N], B[N][N], C[N][N];
/*
initialize arrays
*/
#pragma omp target
{
for (int i = 0; i < N; ++i) {
for (int j = 0; j < N; ++j) {
C[i][j] = A[i][j] + B[i][j];
}
}
} // end target
```



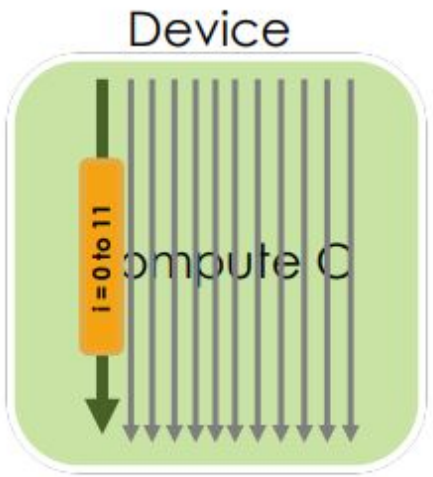
Device Execution Directives

C/C++	Description
#pragma omp target [clause[[,] clause] ...] new-line structured-block	The target construct offloads the enclosed code to the accelerator.
#pragma omp target teams [clause[[,] clause] ...] new-line structured-block	The target construct offloads the enclosed code to the accelerator.The teams construct creates a league of teams. The initial thread of each team executes the code region.
#pragma omp target teams distribute [clause[[,] clause] ...] new-line loop-nest	The target construct offloads the enclosed code to the accelerator. A league of thread teams is created, and loop iterations are distributed and executed by the initial teams.
#pragma omp target teams distribute parallel for [clause[[,] clause] ...] new-line loop-nest	The target construct offloads the enclosed code to the accelerator. A league of thread teams are created, and loop iterations are distributed and executed in parallel by all threads of theteams.

Device Execution Directives(Cont.)

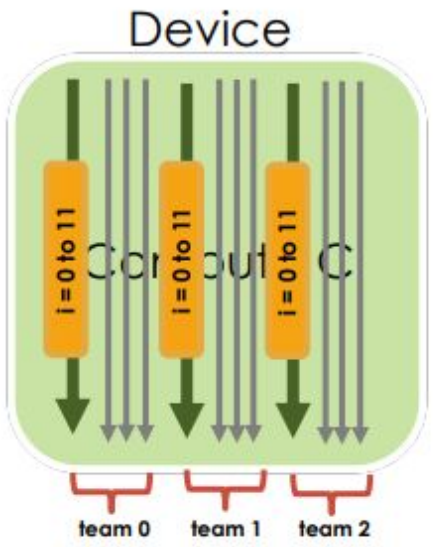
target

```
#pragma omp target
for (int i = 0; i < 12; ++i)
{
    C[i] = A[i] + B[i];
}
```



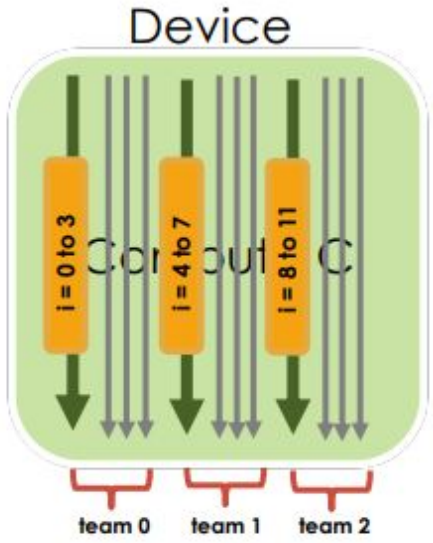
target teams

```
#pragma omp target teams
num_teams(3)
for (int i = 0; i < 12; ++i)
{
    C[i] = A[i] + B[i];
}
```



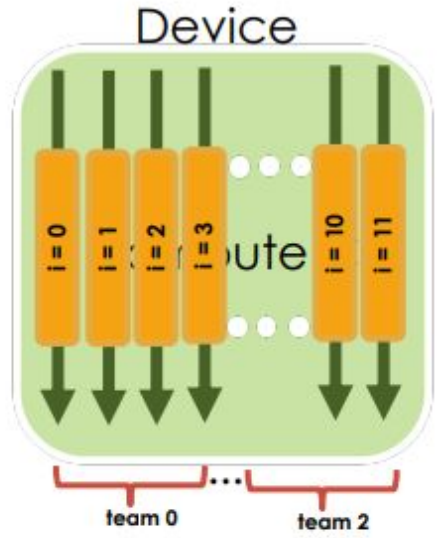
target teams distribute

```
#pragma omp target teams
distribute num_teams(3)
for (int i = 0; i < 12; ++i)
{
    C[i] = A[i] + B[i];
}
```



target teams distribute parallel

```
#pragma omp target teams
distribute parallel for
num_teams(3)
for (int i = 0; i < 12; ++i)
{
    C[i] = A[i] + B[i];
}
```



Demo

Thank You