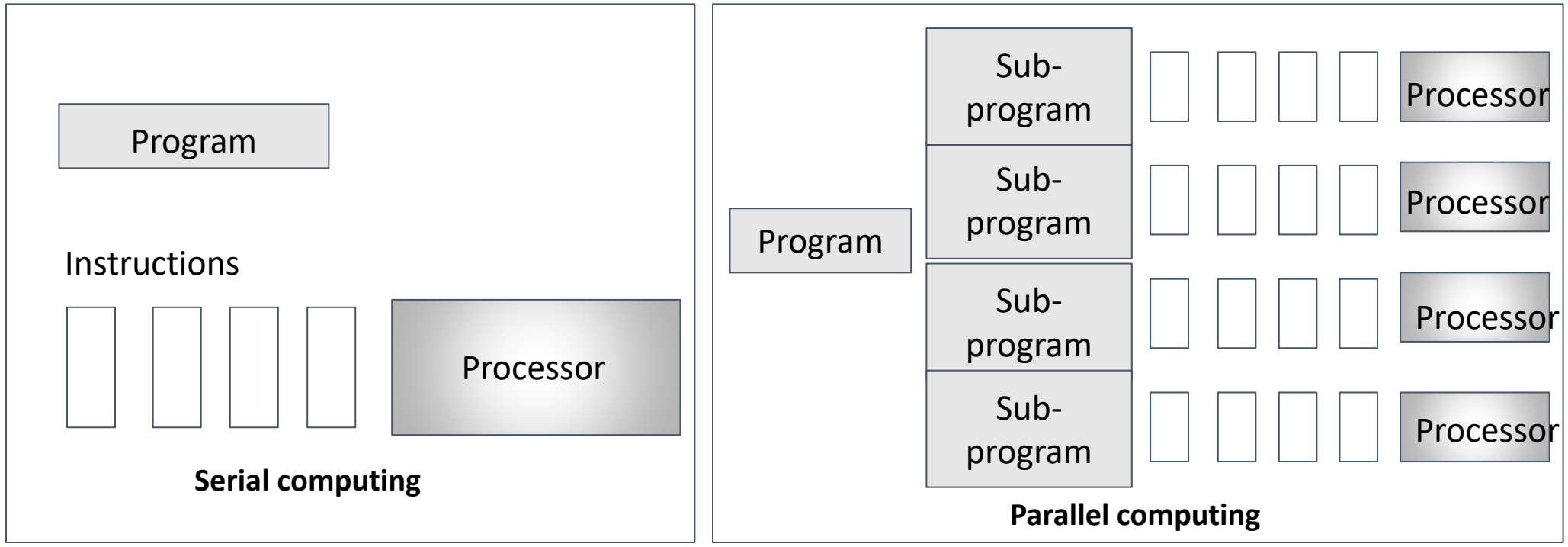
Introduction to Parallel Programming

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?



Introduction to parallel programming

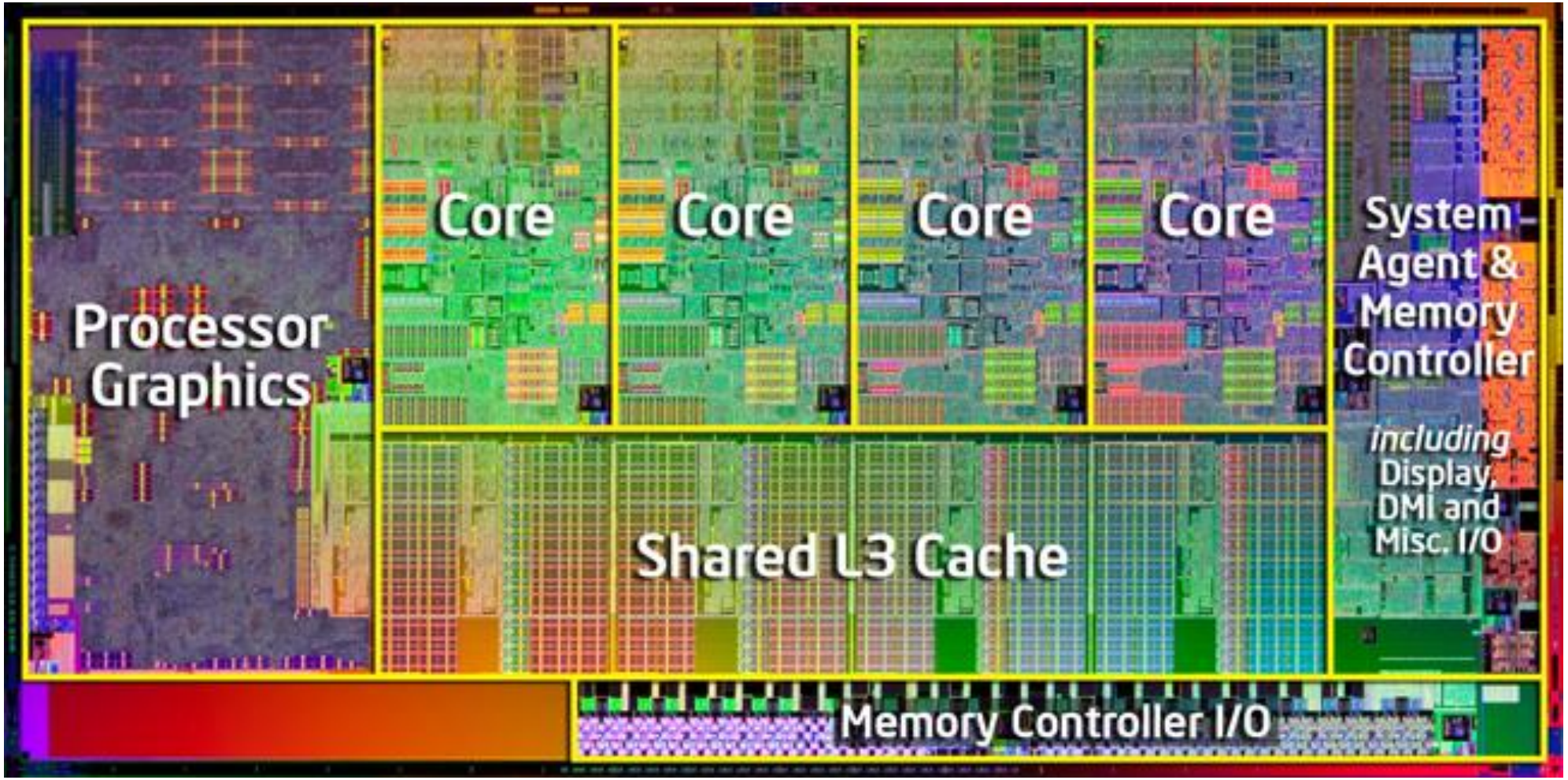
- Need of parallel programming.
  - The exponential growth of processing and network speeds means that parallel architecture isn’t just a good idea; it’s necessary. Big data and the IoT will soon force us to crunch trillions of data points at once.
  - Dual-core, quad-core, 8-core, and even 56-core chips 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
- LEVERAGE REMOTE RESOURCES

Introduction to parallel hardware.



General Parallel Computing Terminology

- 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.

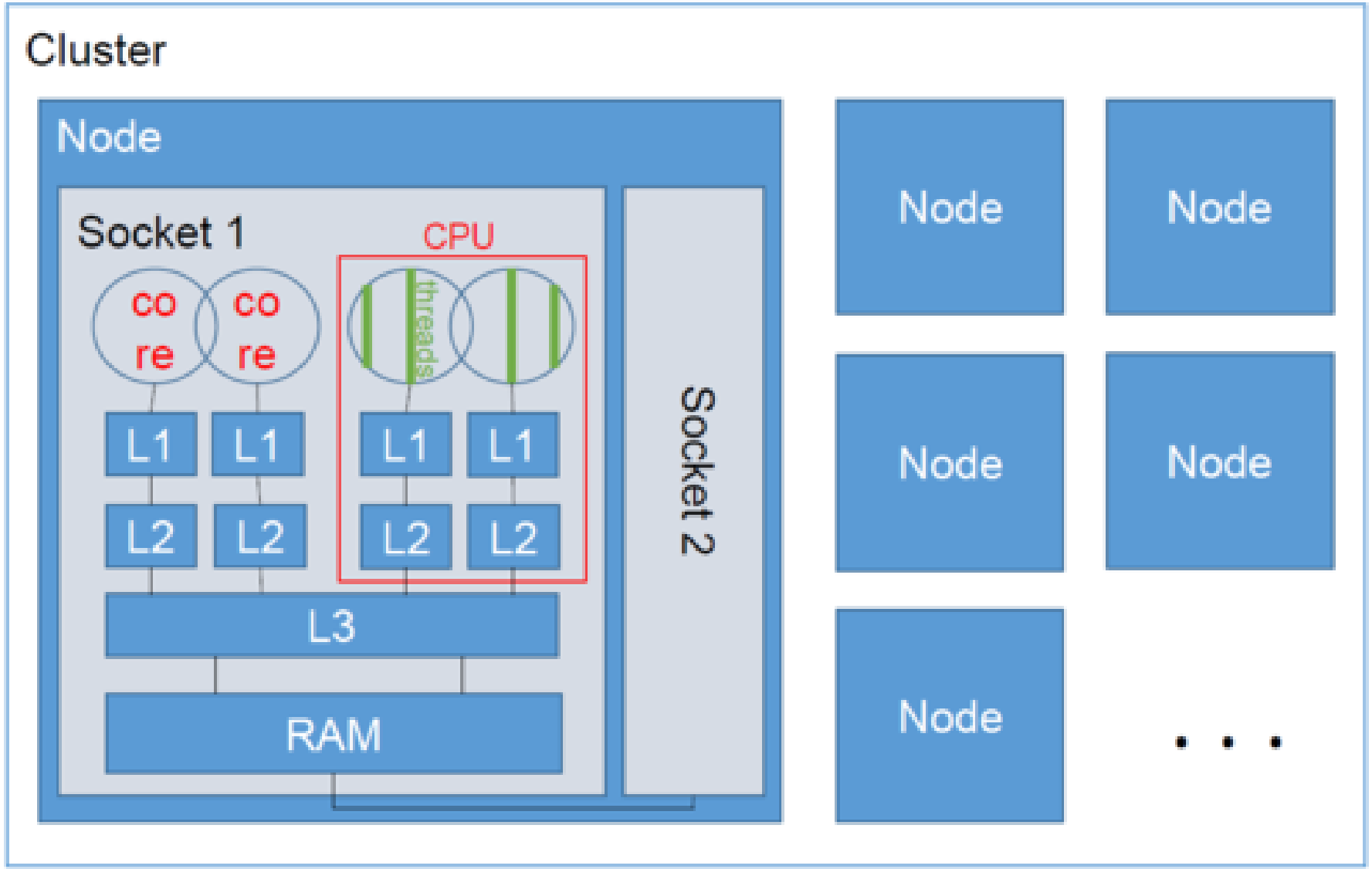
- Process

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

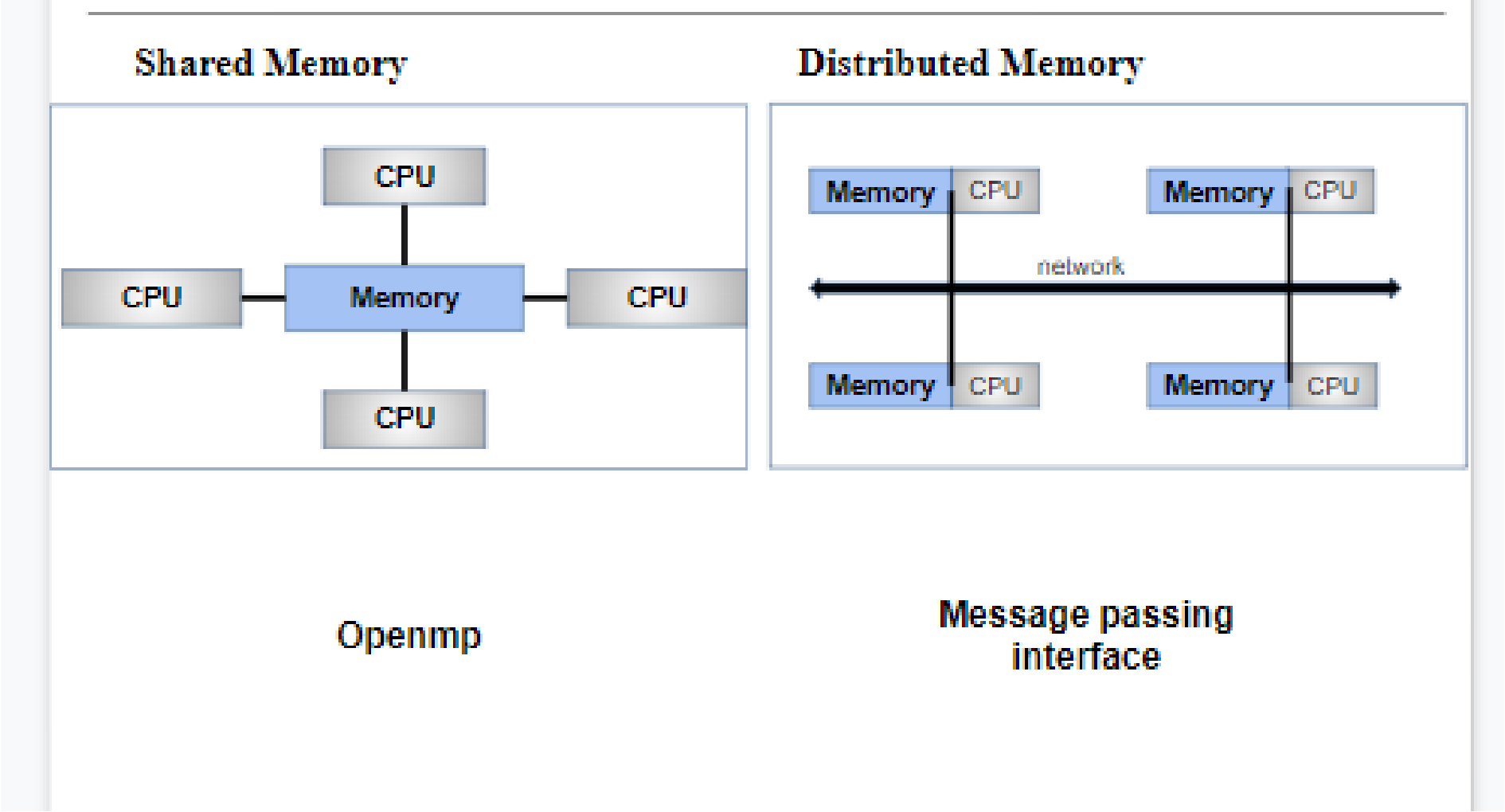
- Thread

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

General Parallel Computing Terminology

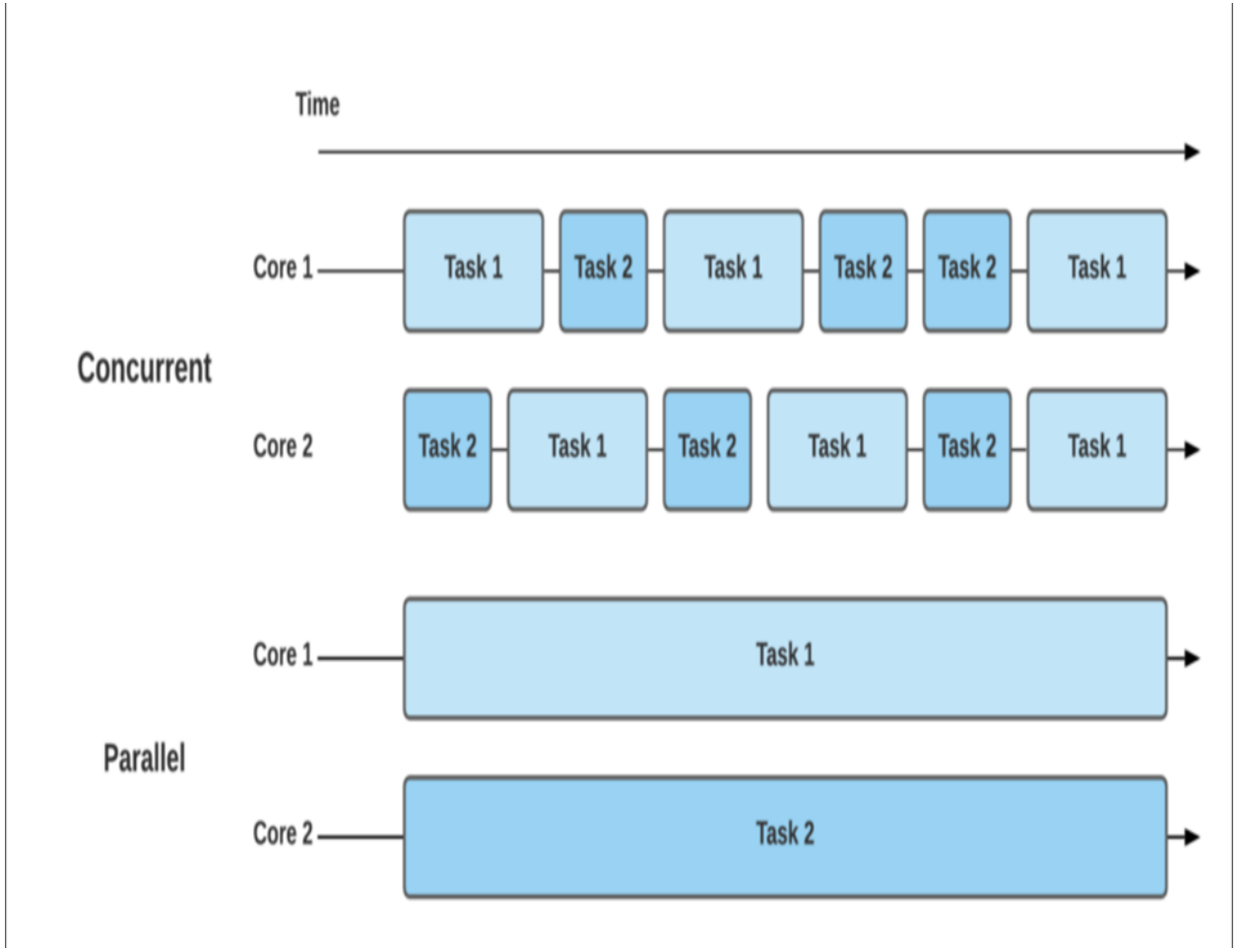


General Parallel Computing Terminology



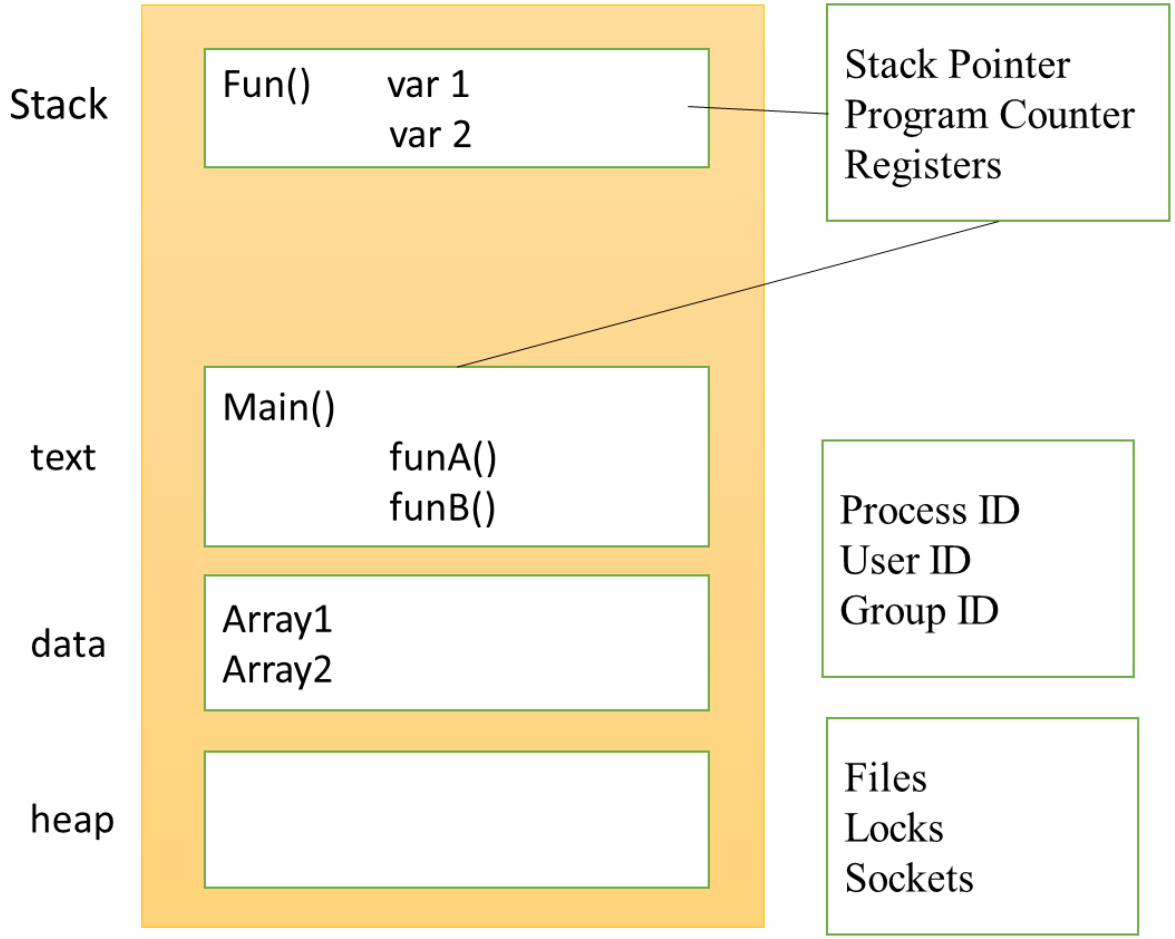
General Parallel Computing Terminology

- 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



Process and Threads

- How program execute in memory?



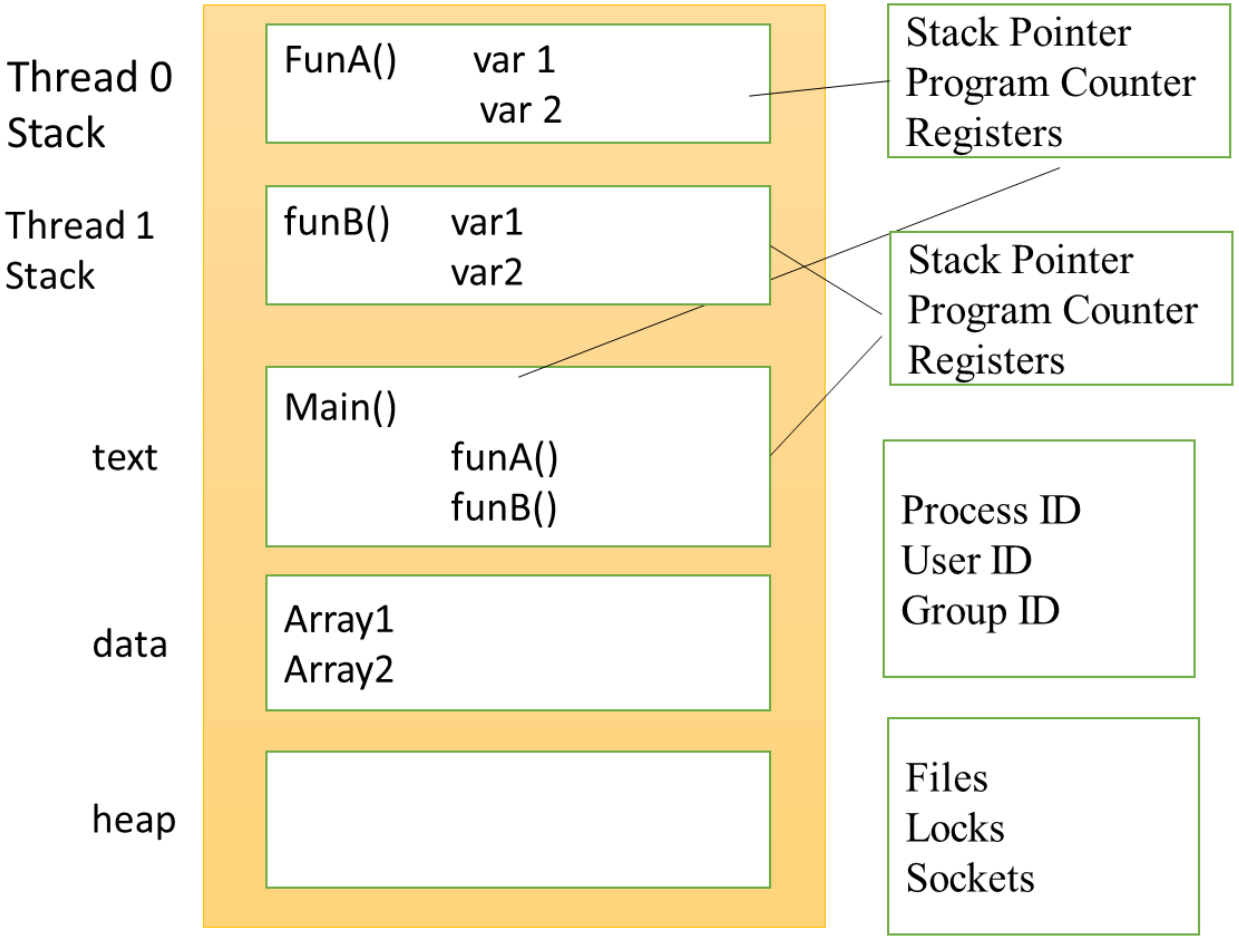
Process:

- An instance of a program execution.

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

Process and Threads

- 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>

Thank you

Introduction to OPENMP

Open Multi Processing

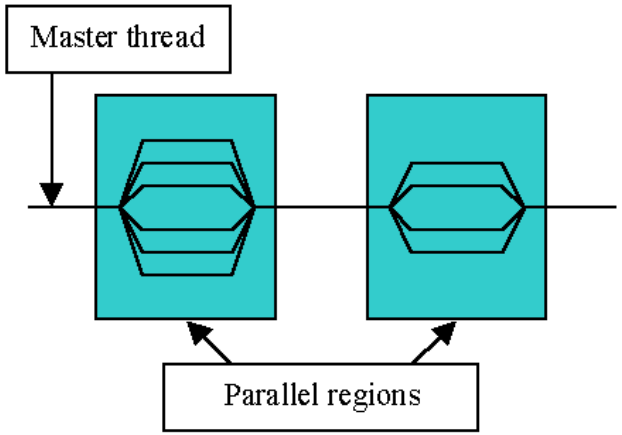
Content

- 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

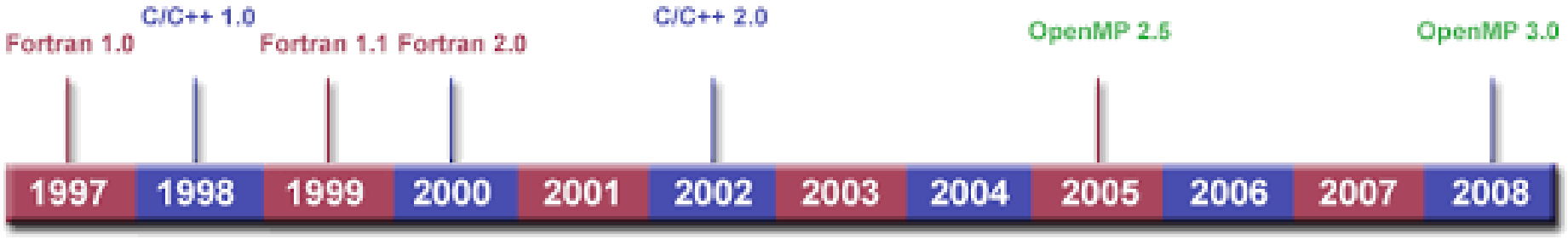
Introduction to openmp

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



Version 1.0 : OpenMP for Fortran 1.0, in October 1997

Version 1.0 : OpenMP for C/C++, in October 1998

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

Introduction to openmp

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?

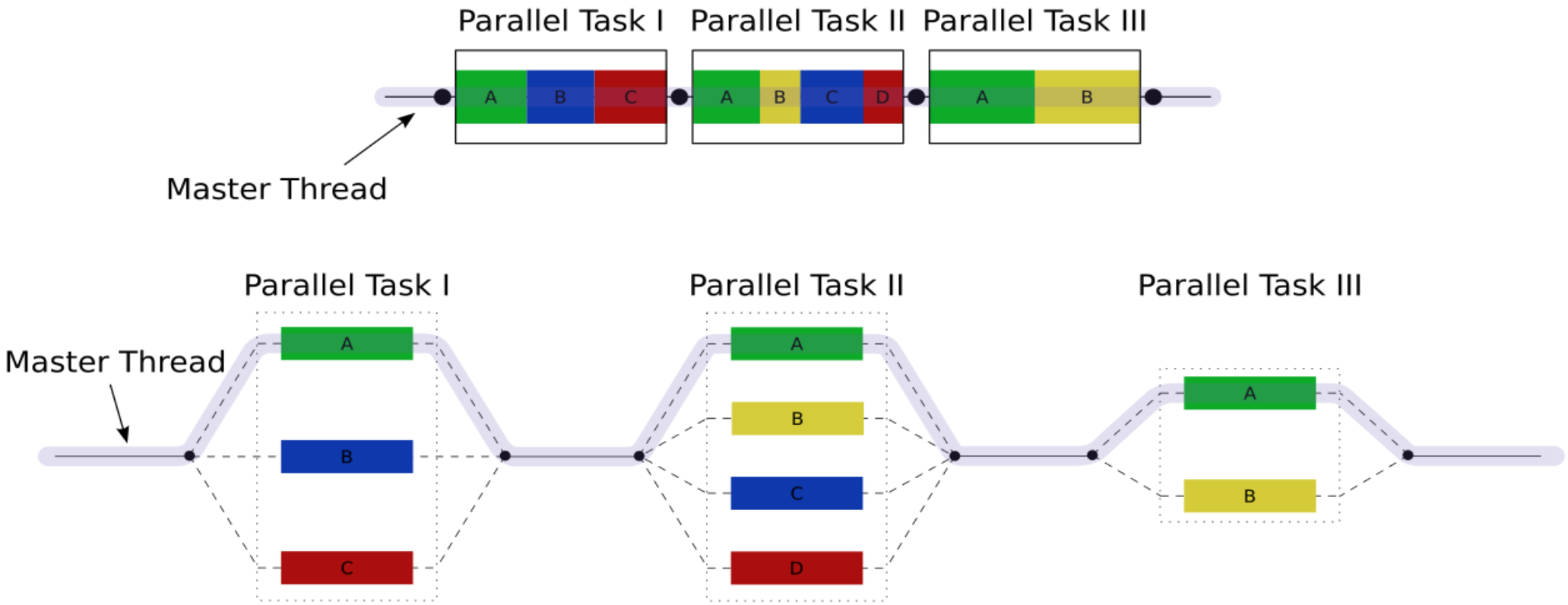
- More efficient
- Hides the low-level details.
- OpenMP has directives that allow the programmer to:
  - specify the parallel region
  - specify whether the variables in the parallel section are private or shared
  - specify how/if the threads are synchronized
  - specify how to parallelize loops
 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

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:

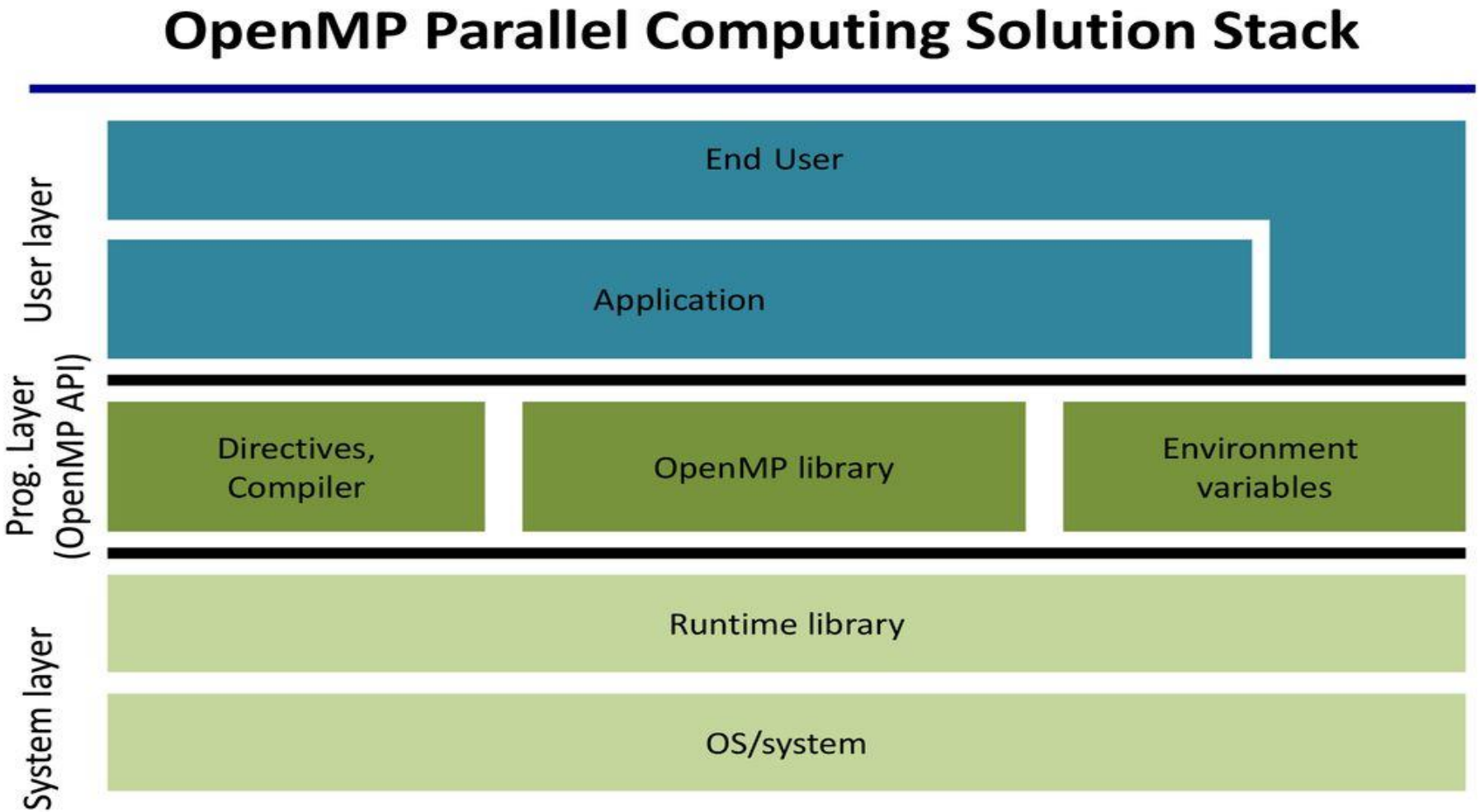


OpenMP Programming Model

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.

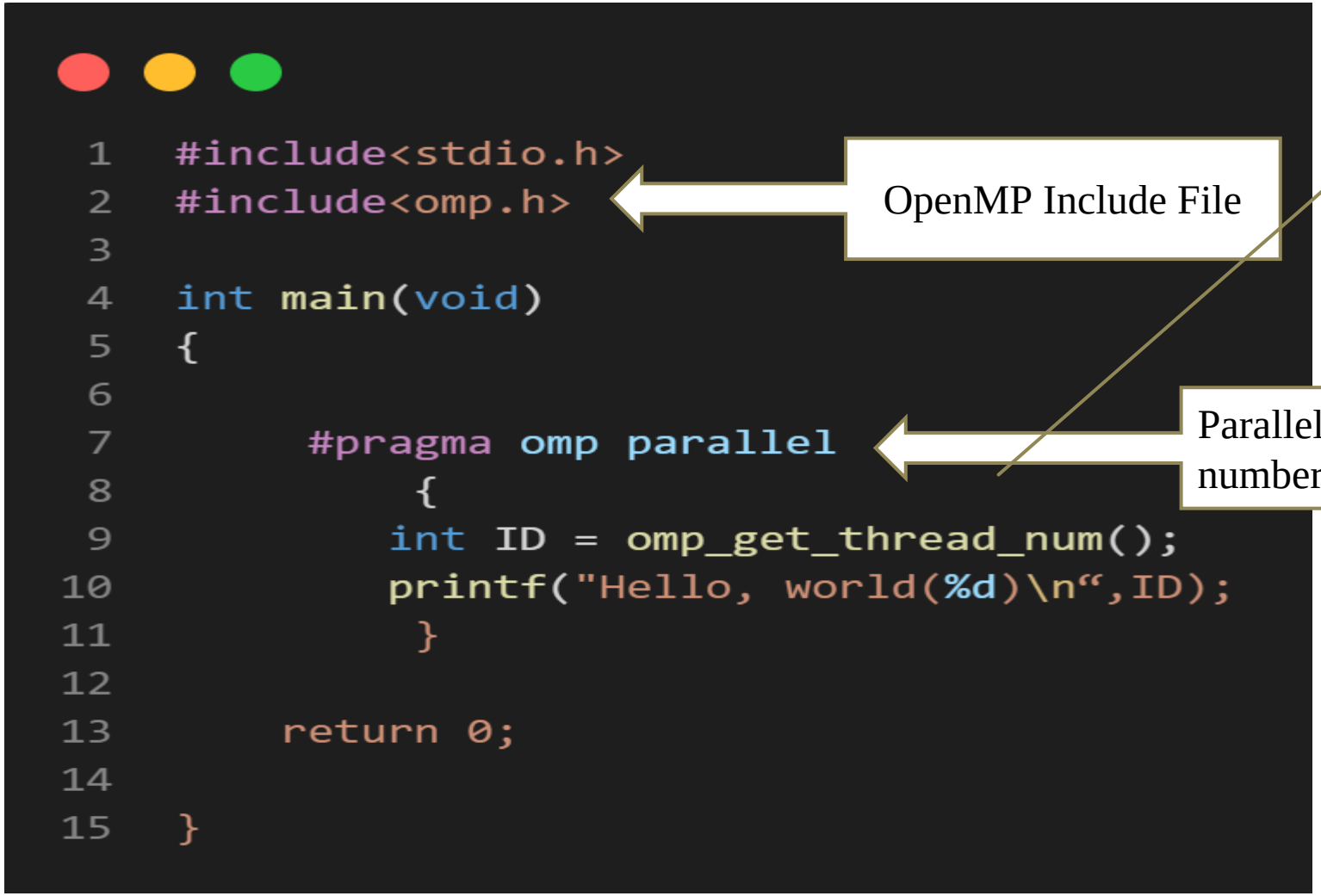
```
#pragma omp construct [clause [clause]…]
        {
        //..Do some work here
        }
        //end of parallel region/block
```

- Example:

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

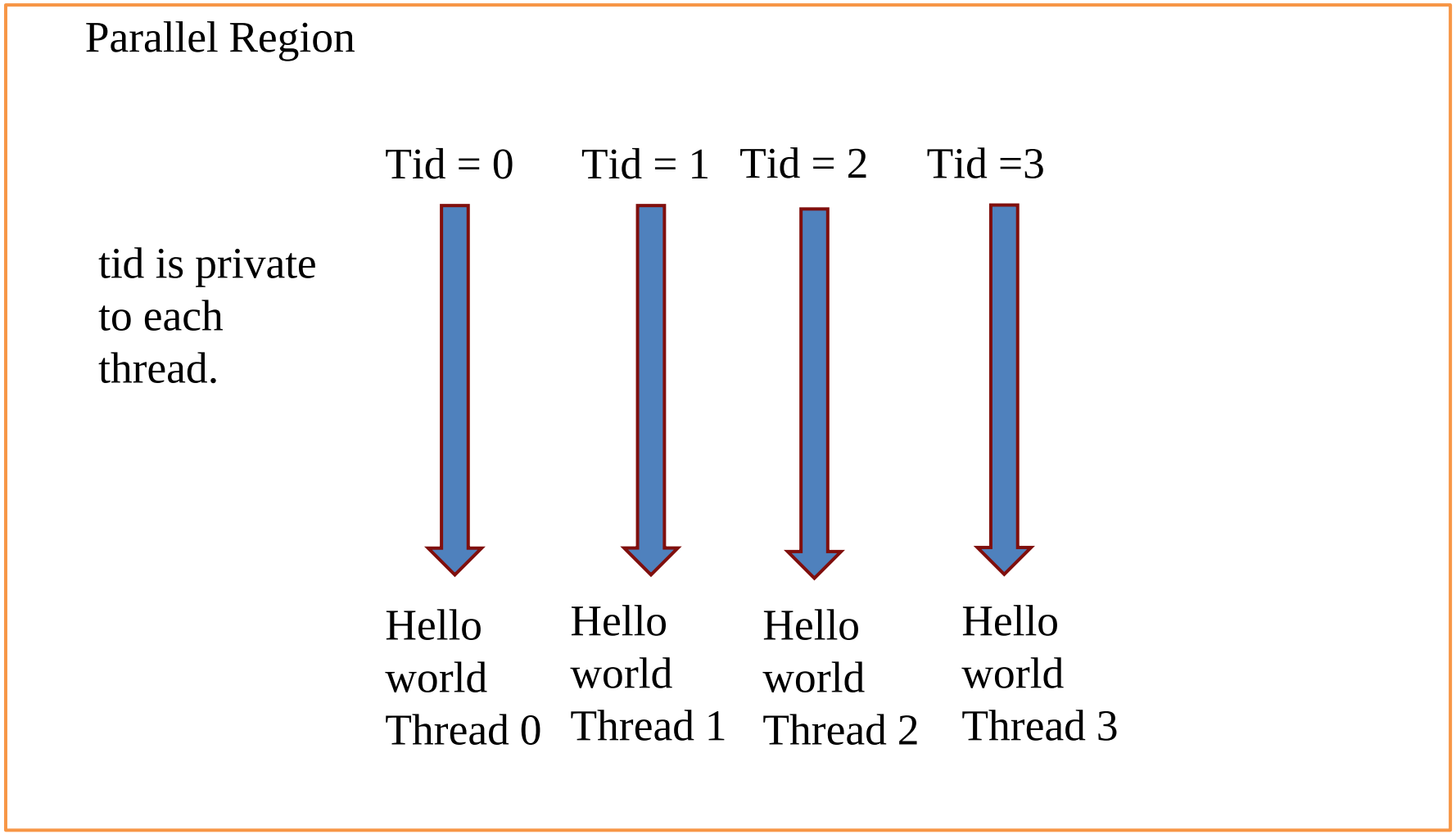
Hello world in openmp

OpenMP Hello World program using C



Hello world in openmp

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

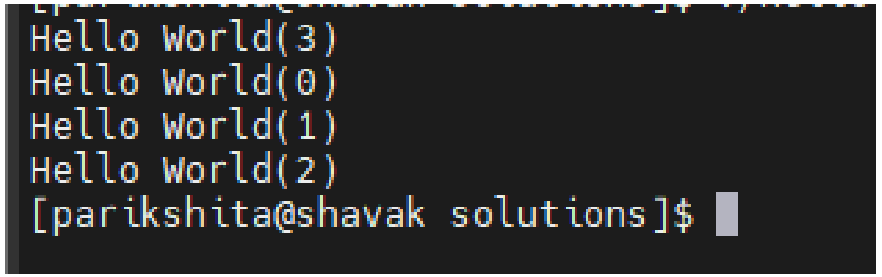
  $./Hello

- For Intel compiler:

  $ icc -qopenmp HelloWorld.c -o Hello

  $./Hello

Output of hello world program with 4 threads:




THANK YOU
FOR YOUR ATTENTION

OpenMP Components

Language extensions

Content

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

OpenMP Components

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
```

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])

collapse(n)

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?

  Example: consider we have 2 threads and N iterations then

  Threads 0 - 0 - N/2-1

  Threads 1 - N/2 - 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 behavior 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;
}
```

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;
}
```

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

```
$ ./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.
```

For loop - SCHEDULE(dynamic)

schedule(dynamic, [n])

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

```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_thread_num(), i);
    }
    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_thread_num(), i);
    }
    return 0;
}
```

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

- Default chunk size is 1.

```
$ ./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.
```

```
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.
```

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.
- 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)

```
$ ./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.
```

```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;
}
```

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

```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 varable a,b,c,chuk */
#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;
}

```

```
$./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
```

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

SECTIONS - FIRSTPRIVATE

```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 varable a,b,c,chuk */
#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[i],d[i]);
    }
    }  /* end of sections */
  }  /* end of parallel section */
 return 0;
}
```

```
$ ./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
```

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

SECTIONS - LASTPRIVATE

```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 havin
#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

```
$ ./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 variable and shared varable a,b,c,chuk */
#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
#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

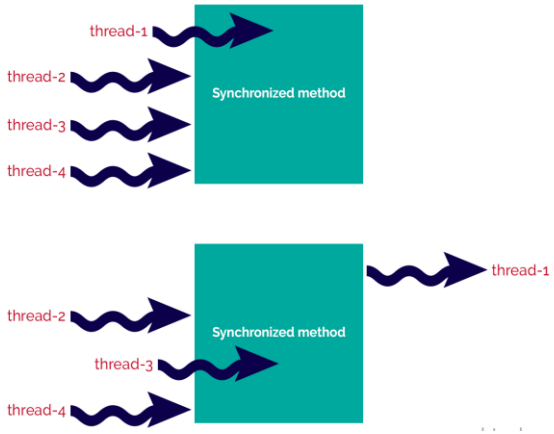
- What happens if the number of threads and the number of SECTIONs are different? What if there are more threads than SECTIONs? fewer threads than SECTIONs?
- Which thread executes which SECTION?

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

```
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>

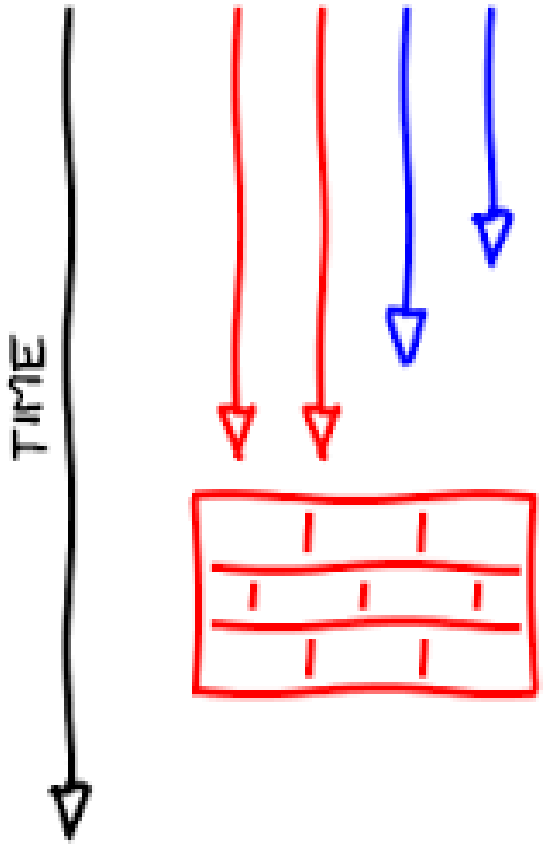
```
$ ./Master
[Thread 0] Every thread executes this printf.
[Thread 3] Every thread executes this printf.
[Thread 1] Every thread executes this printf.
[Thread 2] Every thread executes this printf.
[Thread 0] Only the master thread executes this printf, which is me.
```

Master Construct

```
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

- 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
- It is illegal to branch into or out of an ORDERED block.
- 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

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.

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)

REDUCTION Clause Example

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

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.

OpenMP Environment Variables

- 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