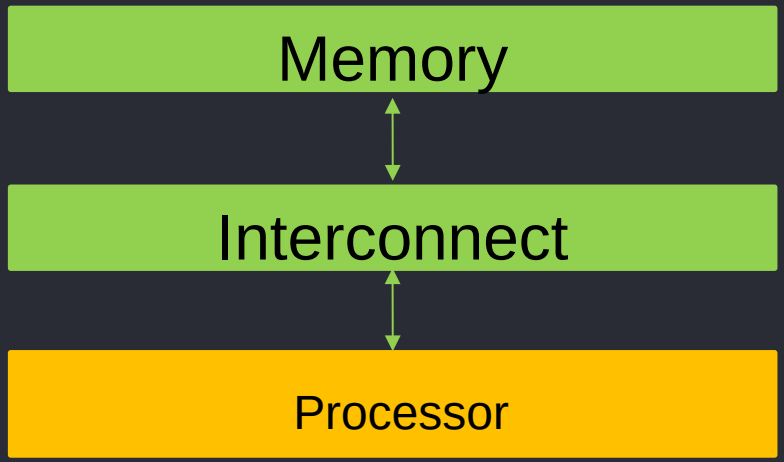
Shared and Dsitrbuted Memory Programming

Dr. Hrachya Astsatryan,
Institute for Informatics and Automation Problems,
National Academy of Sciences of Armenia,
E-mail: hrach@sci.am

INTRODUCTION TO THREADING

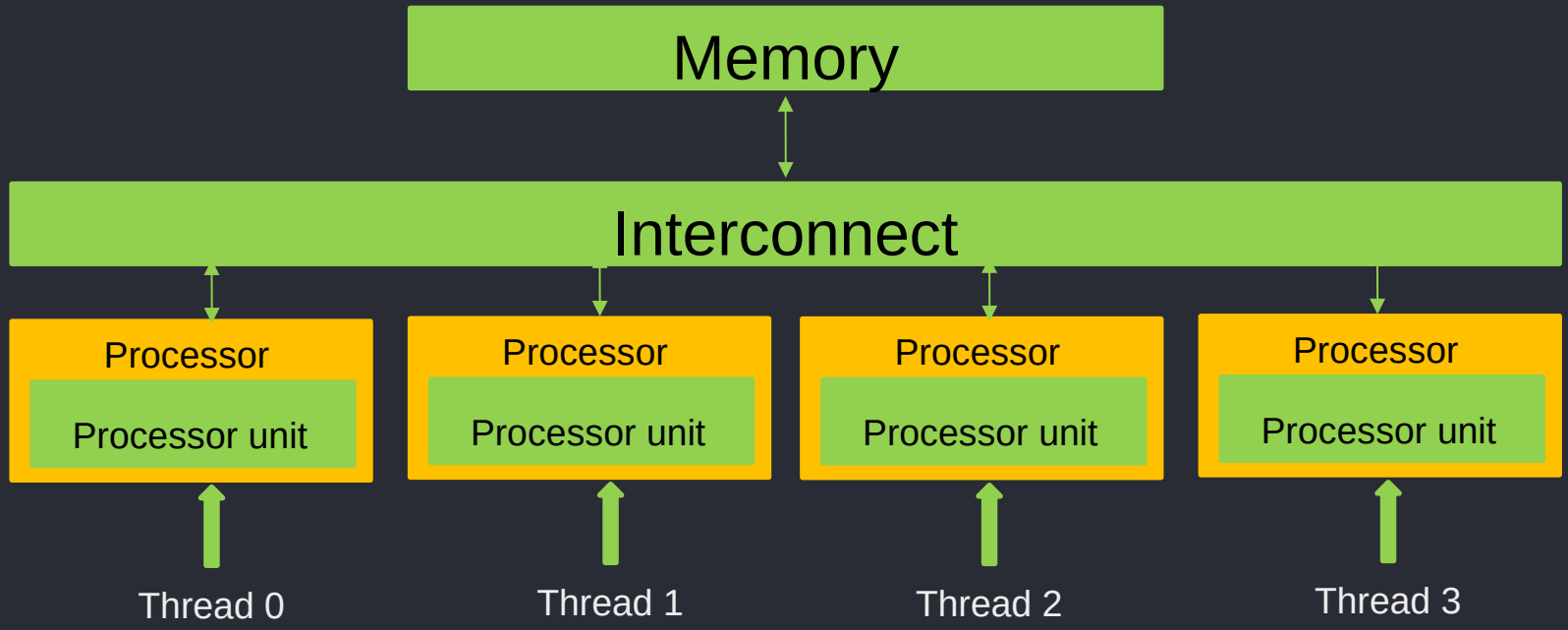
Sequential Programming

Single processor has access certain amount of memory.



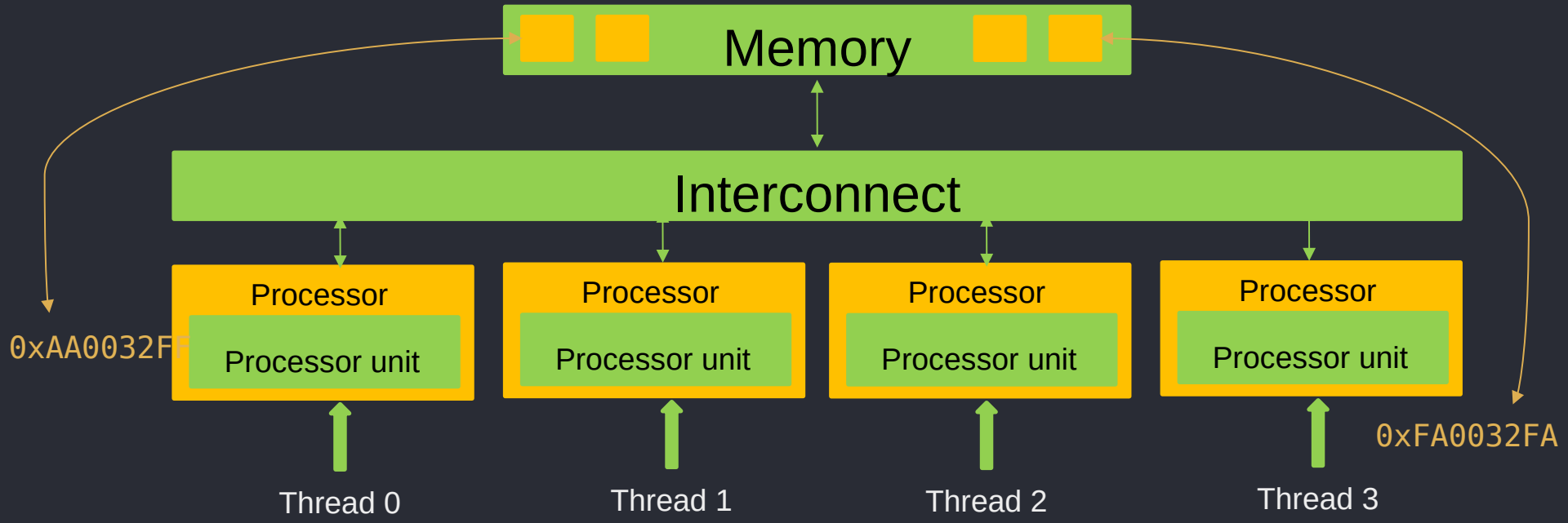
Shared Memory Programming

Multiple CPUs/cores share access to a global memory space via a high-speed memory bus. This global memory space allows the processors to efficiently exchange or share access to data.



Shared Memory Programming

Multiple CPUs/cores share access to a global memory space via a high-speed memory bus. This global memory space allows the processors to efficiently exchange or share access to data.



Hyperthreading

Simultaneous multithreading (SMT) and Hyper-Threading (HT) are technologies that enable more efficient utilization of processor resources by allowing multiple threads to run on a single physical core simultaneously.

The ability to switch between these threads creates an illusion of simultaneous execution, although at any given time the CPU is processing only one thread per core.

Technologies like Intel's Hyper-Threading allow the processor to issue instructions from multiple threads in a single cycle requiring complex control logic to manage instruction dependencies and allocate resources among threads.

Hyperthreading: pros & cons

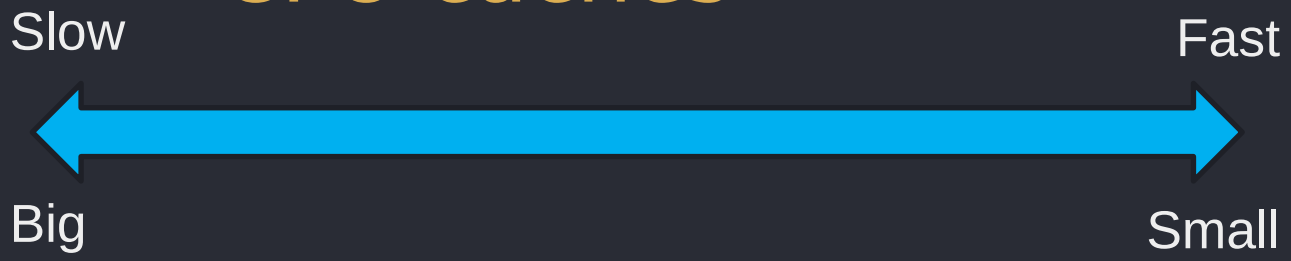
Advantages

- better utilization of CPU resources
- reducing idle time
- handling multiple threads concurrently

Drawbacks

- resource contention often sharing the same hardware resources like caches and translation lookaside buffers. As threads compete for the same resources, potentially causing performance degradation
- More power consumption by keeping additional hardware resources active to support simultaneous thread execution
- Security Concerns such as side-channel attacks, where one thread may exploit shared resources to gain unauthorized access to sensitive information from another thread

CPU caches



During each read query, it is first checked in CPU’s L1 cache, if it is found then returned, else check L2 cache and so on till L3 cache or L4 cache

Memory

L3/4 Cache

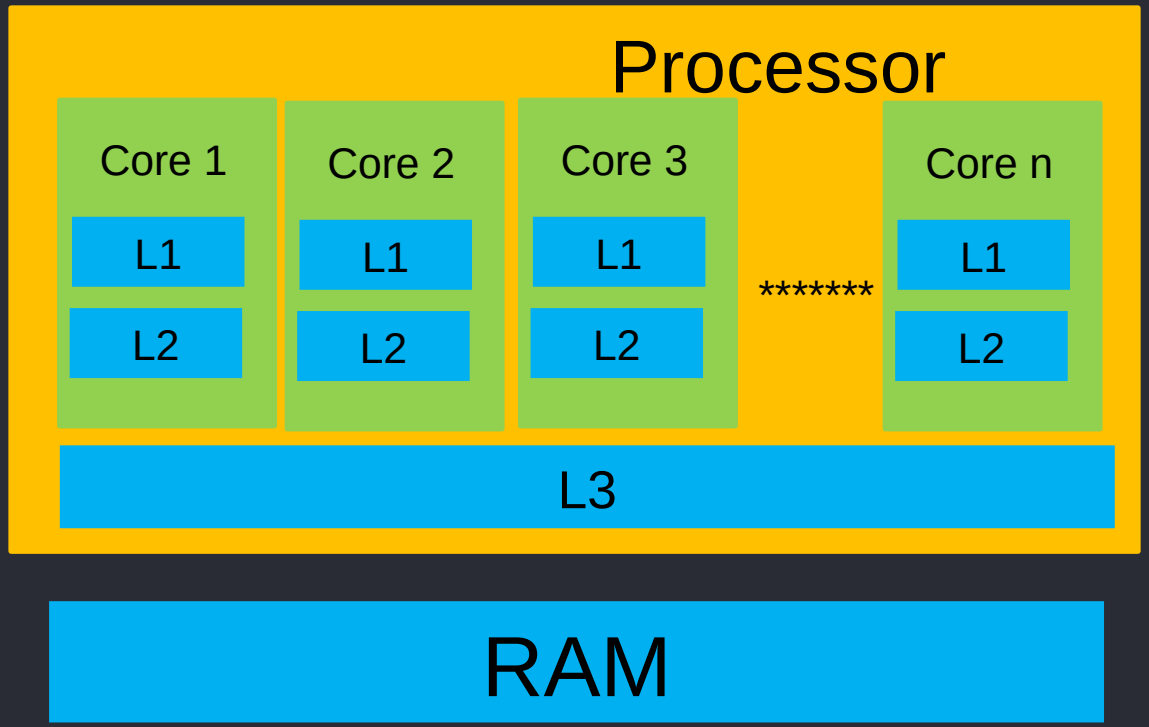
L2 Cache

L1 Cache

CPU

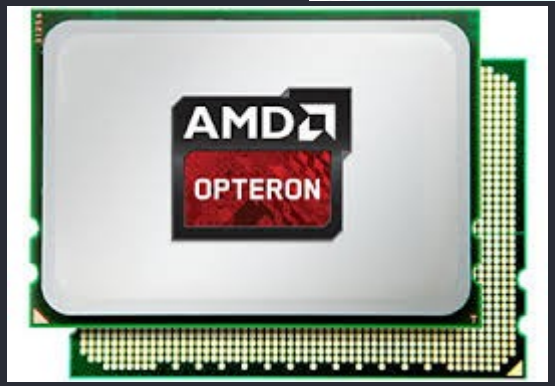
CPU caches

- CPU cache stores frequently accessed data and instructions, enabling the processor to retrieve this information quickly when needed.



Shared Memory programming

- Determine the number of processors:
- cat /proc/cpuinfo
- nproc
- lscpu



2

OpenMP - Open Multi-Processing

Overview

OpenMP is a portable (v. 1.0 1997, v. 5.1 -2020), scalable model and API that gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer.

Produced by a large consortium, suh as AMD, Cray, Fujitsu, HP, IBM, Intel, or NVIDIA

- Languages: C, C++, Fortran, Python, etc.
- OS: Linux, Windows, etc.

Tutorial: computing.llnl.gov/tutorials/openMP

Compilers

- GCC (-fopenmp)
- clang(-fopenmp)/LLVM (Low-Level Virtual Machine)
- Intel Classic and Next-gen Compilers
- AOCC(AMD Optimizing C/C++ Compiler), AOMP (AMD Optimizing Multi-Processing Compiler),
- ROCmCC (Radeon Open Compute C Compiler)
- IBM XL (-qsmp=omp)
- … and many more

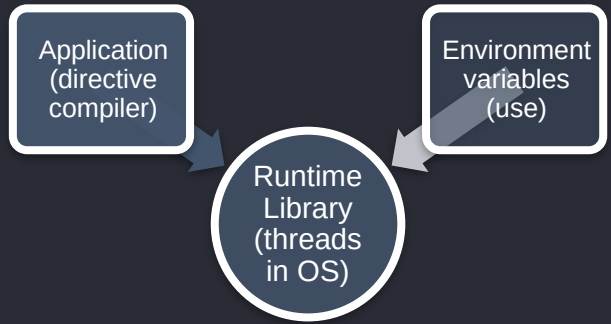
Aims

Standardization
- Standards for variety of shared memory architectures/platforms.

Lean and Mean
- A simple and limited set of directives.

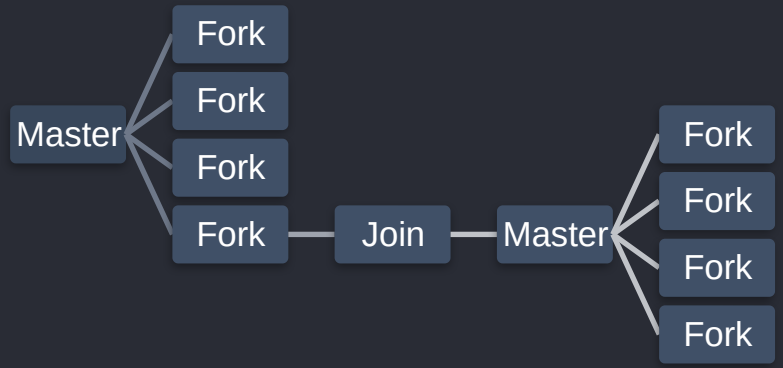
Ease of Use
- Capability to incrementally parallelize a serial program

Portability
- Fortran, C, and C++



Execution Model – Fork/Join

- Initially only master thread is active.
- Master thread executes sequential code.
- Fork: Master thread creates or awakens additional threads to execute parallel code.
- Join: At end of parallel code created threads die or are suspended.



Parallel programming

Matrix product(N×N)

- spawn T threads
- Each thread computes N/T rows of the resultmatrix
- the computations performed by each thread are logically independent from the others

Sum of the elements of a (N×N) matrix

- Result is a scalar
- Spawn T threads
- Each thread computes the sum of N/T rows
- One thread computes the sum of the results of the other threads

Standard C

```
#include <stdio.h>
#include <time.h>
#include <omp.h>
void simulate (int steps)  {
double result = 0.0;
for (int i = 0; i < steps; i++) {
result += i * i;
}
printf("Result: %f\n", result);
}
main() {
int steps = 10000000000;;
double start_time, end_time;
start_time = omp_get_wtime();
simulate(steps);
end_time = omp_get_wtime();
printf("Time with 1 processor: %f seconds\n", end_time - start_time);
}
```

gcc –fopenmp – o single single.c

Syntax

C/C++

- #pragma omp construct [clause [clause] . . .]

F77

- C$OMP construct [clause [clause] . . .]

F90

- !$OMP construct [clause [clause] . . .]

Accessing Library Functions

- C/C++

```
#include <omp.h>
. . .
void omp_set_num_threads (int num_threads);
```

- F77

```
include "omp_lib.h"
. . .
call omp_set_num_threads (num_threads) ;
```

- F90

```
USE omp_lib
. . .
call omp_set_num_threads (num_threads) ;
```

Timing Functions

- C/C++

double omp_get_wtime (void)

double omp_get_wtick (void)

- Fortran

double precision function OMP_GET_WTIME ()

double precision function OMP_GET_WTICK ()

Standard C

```
#include <stdio.h>
#include <time.h>
#include <omp.h>
void simulate (int steps)  {
double result = 0.0;
#pragma omp parallel for reduction(+:result)
for (int i = 0; i < steps; i++) {
result += i * i;
}
printf("Result: %f\n", result);
}
main() {
int steps = 10000000000;;
double start_time, end_time;
start_time = omp_get_wtime();
omp_set_num_threads(2);
simulate(steps);
end_time = omp_get_wtime();
printf("Time with 2 processor: %f seconds\n", end_time - start_time);
}
```

```
gcc –fopenmp – o myexecutable myprogram.c
```

3

OpenMP Environmental Variables

Get Variables

There are 17 different library routines, we will cover just a few of them now.

- int omp_get_max_threads (void) - maximum number of threads that the run-time system will let our program create.
- int omp_get_num_procs (void) - the number of processors the parallel program can use.
- int omp_get_num_threads (void) - the number of threads that are currently active.
- int omp_get_thread_num (void) - retums the thread's identification number. If there are n active threads. the thread identification numbers range from 0 to n-1

Set Variables

- void omp_set_num_threads { int t /* :--Tumber of threads desired */ } - Sets the desired number of parallel threads for subsequent executions of parallel regions. The number of threads may exceed the number of available processors, in which case multiple threads may be mapped to the same processor. This call must be made from a serial portion of a program.

- void omp_ set_dynamic (int k / i = ON, 0 = FALSE / - It used to enable or disable dynamic threads. If dynamic threads are enabled, the run-time system may adjust the number of active threads to the humber of physical processors available.

Get & Set

The value of an environment variable called OMP_NUM_THREADS provides a default number of threads for parallel sections of code.

In Unix script we can write:

export OMP_NUM_THREADS = Number

```
int num_threads;
num_threads = omp_get_num_procs ();
omp_set_num_threads(num_threads);
```

OpenMP Directives

Parallel

#pragma omp parallel - the code inside the region is executed by multiple threads.

```
main() {
int i,n;

for (i=0; i<=n; i++) {
            sum = sum+i;

  }
}
return 0;
}
```

```
#include <omp.h>
main() {

int i,n=10, sum=0;
#pragma omp parallel
{
for (i=0; i<=n; i++) {
            sum = sum+i;
  }
printf ("1\n");
}
return 0;
}
```

Parallel for

- #pragma omp parallel for - parallelizing loops. It allows the iterations of a loop to be executed concurrently across multiple threads, distributing the workload among the available processor cores

```
main() {

int i;
for (i=0; i<=n; i++) {
            sum =
sum+i;

  }
return 0;
}
```

```
#include <omp.h>
main() {

int i,n=10, sum=0;
#pragma omp parallel for
for (i=0; i<=n; i++) {
            sum = sum+i;
  }
printf (“1\n”);

return 0;
}
```

Loop scheduling - static

#pragma omp parallel for schedule(static[, chunk]) - divides the iteration space of a parallel loop into blocks of a specified size. These blocks are then assigned to threads in a round-robin fashion.

Suppose you have a loop with 100 iterations that you want to parallelize using OpenMP with 4 threads.
If you use a static schedule with a chunk size of 10, each thread will be assigned a block of 10 iterations statically. So, thread 1 will handle iterations 0-9, thread 2 will handle iterations 10-19, and so on. This allocation is fixed at the beginning and remains the same throughout the loop.

Loop scheduling - dynamic

#pragma omp parallel for schedule(dynamic[, chunk]) - divides the iteration space into blocks of a specified size (or 1 if not specified). These blocks are dynamically scheduled to threads in the order in which threads finish processing previous blocks. This means that threads may receive new blocks of iterations as they become available, allowing for better load balancing.

With a dynamic schedule and a chunk size of 10, the iterations will be dynamically distributed among the threads. Initially, each thread might receive a block of 10 iterations. But as threads finish their work, they will request more work from the pool of remaining iterations. So, if thread 1 finishes its block early, it might request another block of iterations to work on, and so forth.

Loop scheduling - guided

#pragma omp parallel for schedule(guided[, chunk]) - similar to dynamic schedule, but the size of the blocks decreases dynamically over time. Initially, larger blocks are assigned to threads, but as the computation progresses, the block size decreases. This can be useful for balancing load in situations where the workload per iteration may vary.

In guided scheduling, the block size decreases over time. Initially, larger blocks of iterations are assigned to threads, but as threads finish their work, the block size decreases. For example, if the initial chunk size is 50, thread 1 might get iterations 0-49, thread 2 might get iterations 50-99, but as threads finish their work, subsequent blocks might be smaller, like 25 or 10 iterations each.

Barrier

#pragma omp barrier – useful for I/O, memory allocation and deallocation, implementation of the single-creator parallel-executor pattern

- Threads wait until all threads of the current Team have reached the barrier
- All work sharing constructs contain an implicit barrier at the end

```
#pragma omp barrier
 printf("Thread %d: After barrier, sum = %d\n", thread_id,
```

Critical

- #pragma omp critical - specifies a critical section of code that can only be executed by one thread at a time. Used to avoid race conditions.

```
#pragma omp parallel {
  int thread_sum = 0;
  #pragma omp for for (int i = 0; i < 10; ++i) {
    thread_sum += i;
} // Critical section ensures that only one thread at a time can update the shared 'sum' variable
#pragma omp critical
sum += thread_sum;
}
```

Atomic

- #pragma omp atomic - specifies that a variable update should be performed atomically, without interference from other threads. It's typically used for simple operations like incrementing or updating a shared variable.

```
#pragma omp parallel {
  #pragma omp for for (int i = 0; i < 10; ++i) {
     #pragma omp atomic
    sum += i;
   }
}
```

Section and Sections

- #pragma omp section - Specifies a section of code in a sections directive.
- #pragma omp sections - divides the enclosed code into sections, each of which is executed by one thread.

```
#pragma omp parallel {
   #pragma omp sections {
     // Section 1: executed by one thread
    #pragma omp section {}
    // Section 2: executed by another thread
   #pragma omp section {}
}
```

Single

- #pragma omp single

imposes that only one of the existing threads performs the following computation (impossible to choose which one though)

Reduction

- #pragma omp reduction - Specifies a reduction operation for a variable in a parallel loop. Reduction operations typically involve combining values from multiple threads into a single result. Common reduction operations include summing elements of an array, finding the maximum or minimum value, or performing bitwise operations like bitwise AND or OR.

```
#pragma omp parallel{
    #pragma omp for reduction(operator:variable)
    for (int i = 0; i < n; ++i) {
        // Loop body
    }
}
```

5

Data sharing

Shared vs private

- **Shared:** Shared data is a single copy in memory that all threads can access. This allows for data to be visible and accessible to all threads, enabling them to read and write to the same memory locations.

- **Private:** Private data is data that is unique to each thread, with each thread having its own copy of the data. This data is placed in the thread's stack, so other threads cannot access it. As a result, the same private data can hold different values for each thread since each thread works with its own isolated copy.

Implicit rules

- **Shared** – declared outside of a parallelsection (e.g. n)
- **Private -** loop iteration variables are automatically cast as private by the compiler (e.g. i)
- **Private –** declared inside a parallelsection are private(e.g. sum)

```
int i=0;
Int n=10;

#pragma omp parallel for
for (i=0; i<=n; i++) {
        int  sum = sum+i;

  }
```

Shared vs private

- **shared(<variables list>) clause**

```
#pragma omp parallel shared(a)
{
... // a is shared by all threads
}
```

- **private(<variables list>) clause**

```
#pragma omp parallel private(b)
{
...  // each thread has its own uninitialized copy of b
}
```

Book References

https://www.openmp.org/resources/openmp-books/