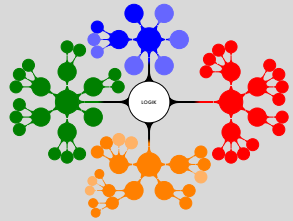
Lecture 10

Programming Shared Memory II

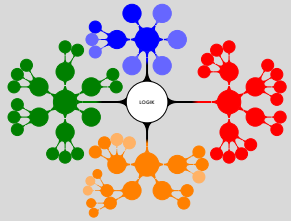
OpenMP (Open Multi-Processing)

IKC-MH.57 *Introduction to High Performance and Parallel Computing* at December 22, 2023

Dr. Cem Özdoğan
Engineering Sciences Department
İzmir Kâtip Çelebi University

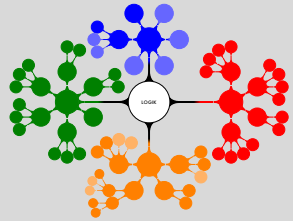


Contents



OpenMP: a Standard for Directive Based Parallel Programming I

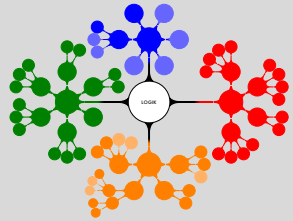
- Although standardization and support for the threaded APIs has a considerable progress, their use is still restricted to **system programmers** as opposed to **application programmers**.
- One of the reasons for this is that APIs such as Pthreads are considered to be **low-level primitives**.
- A large class of applications can be efficiently supported by **higher level constructs** (**or directives**)
- Which rid the programmer of the mechanics of manipulating threads.
- Such **directive-based languages** have standardization efforts succeeded in the form of OpenMP.
- **OpenMP is an API that can be used with FORTRAN, C, and C++ for programming shared address space machines.**



OpenMP: a Standard for Directive Based Parallel Programming II

- Standard API for defining multi-threaded shared-memory programs.
- Allow a programmer to separate a program into serial regions and parallel regions, rather than concurrently-executing threads.
- NOT parallelize automatically and NOT guarantee speedup.
- General structure:

```
1  #include <omp.h>
2  main ()  {
3  int var1, var2, var3;
4  Serial code
5  Beginning of parallel section. Fork a team of threads
6  Specify variable scoping
7  #pragma omp parallel private(var1, var2) shared(var3)
8    {
9    Parallel section executed by all threads
10   All threads join master thread and disband
11   }
12 Resume serial code
13 }
```

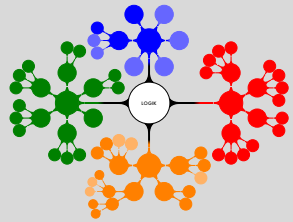



The OpenMP Programming Model I

- OpenMP *directives* provide support for **concurrency**, **synchronization**, and **data handling** while avoiding the need for explicitly setting up mutexes, condition variables, data scope, and initialization.
- OpenMP directives in C is based on the *#pragma* compiler directives.
- The directive itself consists of a directive name followed by clauses.

```
#pragma omp directive [clause list]
```

- OpenMP programs execute serially until they encounter the *parallel* directive.
- This directive is responsible for **creating a group of threads**.
- The exact number of threads can be
  - specified in the directive (num_threads(4)),
  - set using an environment variable (export OMP_NUM_THREADS=4 [sh, ksh, bash]),
  - defined at runtime using OpenMP functions (omp_set_num_threads(4)).



The OpenMP Programming Model II

- The **main** thread that encounters the *parallel* directive becomes the *master* of this group of threads with id 0.
- The *parallel* directive has the following prototype:

```
#pragma omp parallel [clause list]
/* structured block */
```

- Each thread created by this directive executes the *structured block* specified by the parallel directive (SPMD).

```
1 int main() {
2     omp_set_num_threads(4);
3     // Do this part in parallel
4     #pragma omp parallel
5     {
6         printf( "Hello, World!\n" );
7     }
8 return 0;
```

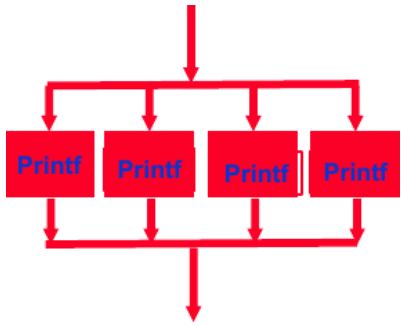
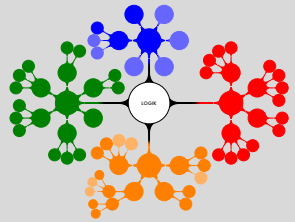


Figure: Creating four threads for "printf" function.



The OpenMP Programming Model III

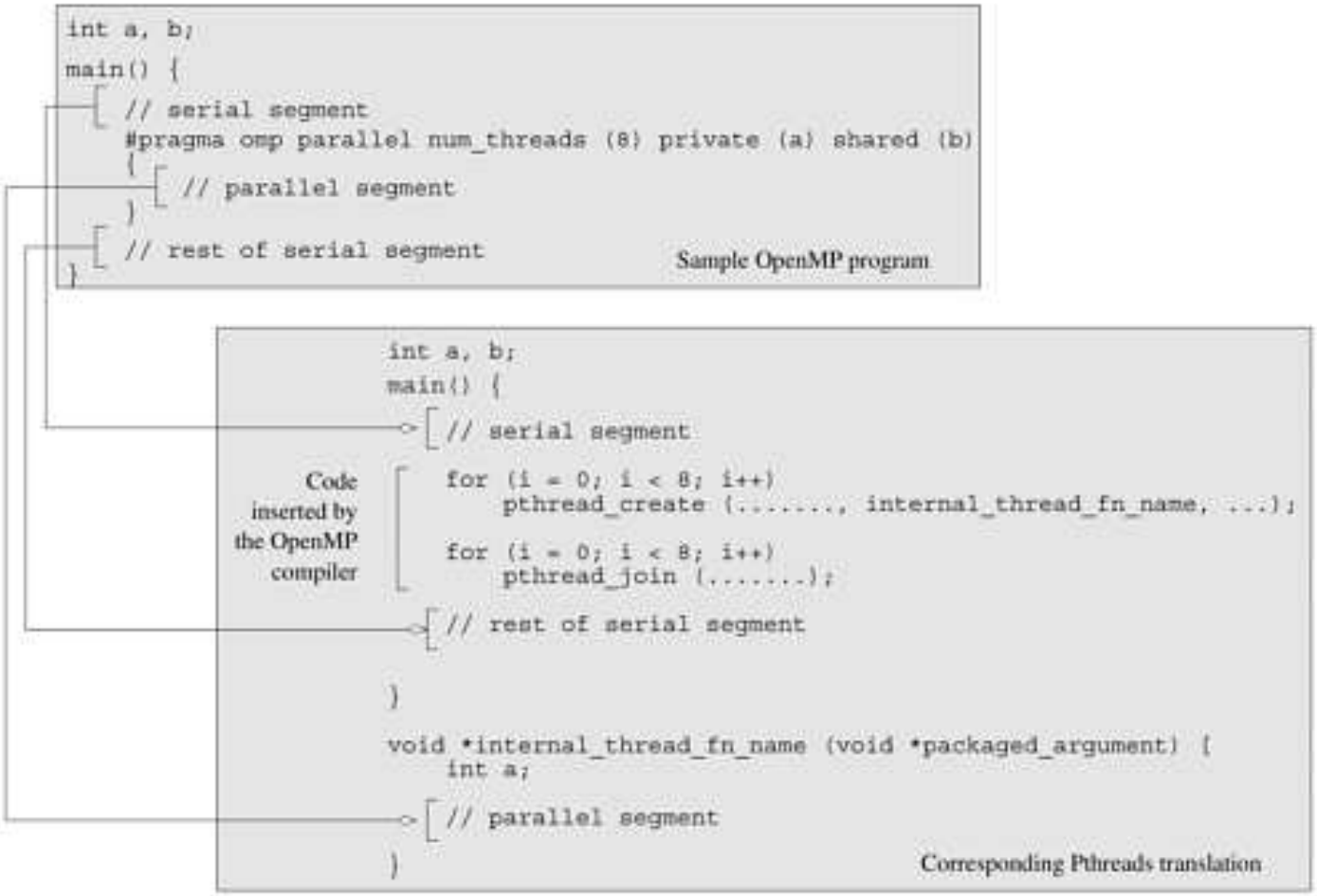
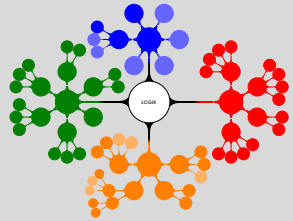


Figure: A sample OpenMP program along with its Pthreads translation that might be performed by an OpenMP compiler.

The OpenMP Programming Model IV

- The clause list is used to specify **conditional parallelization**, **number of threads**, and **data handling**.

1. **Conditional Parallelization:** The clause *if (scalar expression)* determines whether the parallel construct results in creation of threads.
2. **Degree of Concurrency:** The clause *num_threads (integer expression)* specifies the number of threads that are created by the *parallel* directive.
3. **Data Handling:** The clause *private (variable list)* indicates that the set of variables specified is local to each thread.
   - Each thread has its own copy of each variable in the list.
   - The clause *firstprivate (variable list)* is similar to the private clause, except the values of variables on entering the threads are initialized to corresponding values before the parallel directive.
   - The clause *shared (variable list)* indicates that all variables in the list are shared across all the threads,



The OpenMP Programming Model V

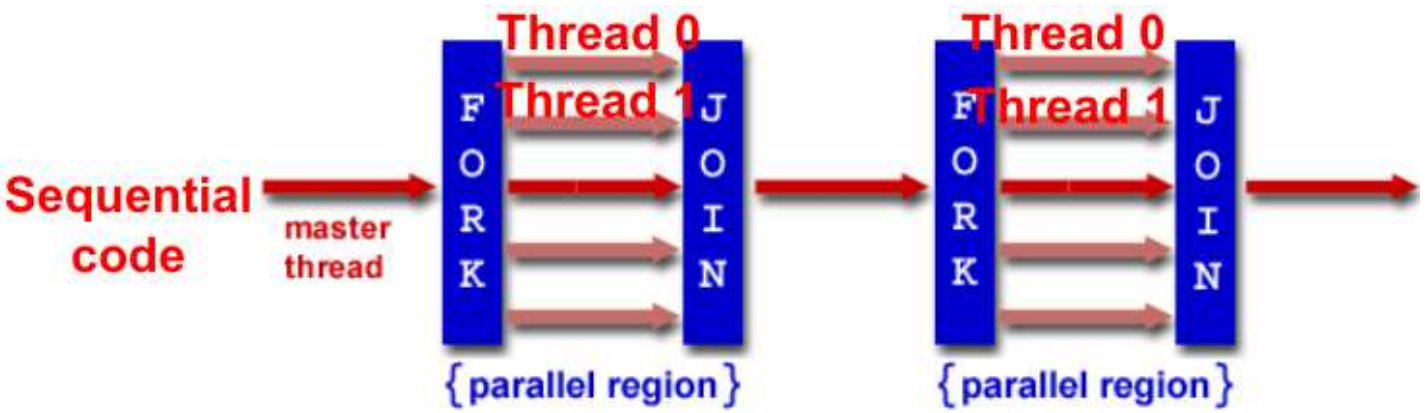
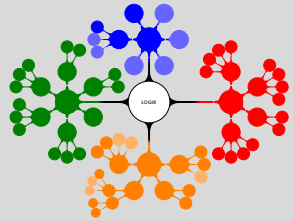


Figure: Fork-Join Model.

FORK Master thread then creates a team of parallel threads.

Statements in program that are enclosed by the parallel region construct are executed in parallel among the various threads.

JOIN When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.

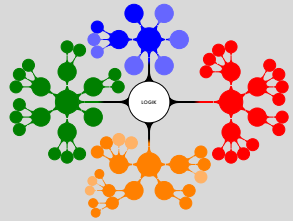


The OpenMP Programming Model VI

Shared Variables. OpenMP default is shared variables. To make private, need to declare with pragma:

```
1  #include <stdio.h>
2  #include <omp.h>
3  #include <unistd.h>
4  int a,b,x,y,num_threads,thread_num;
5  int main()
6  {
7    printf("I am in sequential part.\n");
8  #pragma omp parallel num_threads (8)  private (a) shared (b)
9    {
10     num_threads=omp_get_num_threads();
11     thread_num=omp_get_thread_num();
12     x=thread_num;
13 //      sleep(1);
14     y=x+1;
15     printf("I am openMP parellized part and thread %d. \n X and Y
        values are %d and %d. \n",omp_get_thread_num(),x,y);
16   }
17   printf("I am in sequential part again.\n");
18 }
```

X and y are shared variables. There is a risk of data race.

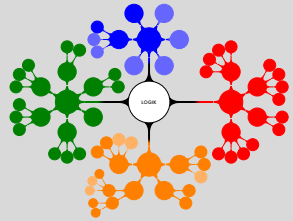


The OpenMP Programming Model VII

```
I am in sequential part.
I am openMP parellized part and thread 2.
     X and Y values are 2 and 3.
I am openMP parellized part and thread 4.
     X and Y values are 4 and 5.
I am openMP parellized part and thread 0.
     X and Y values are 0 and 1.
I am openMP parellized part and thread 1.
     X and Y values are 1 and 2.
I am openMP parellized part and thread 7.
     X and Y values are 7 and 8.
I am openMP parellized part and thread 5.
     X and Y values are 5 and 6.
I am openMP parellized part and thread 3.
     X and Y values are 3 and 4.
I am openMP parellized part and thread 6.
     X and Y values are 6 and 7.
I am in sequential part again.
```

```
I am in sequential part.
I am openMP parellized part and thread 4.
     X and Y values are 5 and 6.
I am openMP parellized part and thread 3.
     X and Y values are 5 and 6.
I am openMP parellized part and thread 1.
     X and Y values are 5 and 6.
I am openMP parellized part and thread 6.
     X and Y values are 5 and 6.
I am openMP parellized part and thread 2.
     X and Y values are 5 and 6.
I am openMP parellized part and thread 5.
     X and Y values are 5 and 6.
I am openMP parellized part and thread 7.
     X and Y values are 5 and 6.
I am openMP parellized part and thread 0.
     X and Y values are 5 and 6.
I am in sequential part again.
```

Table: Correct and Wrong outputs of the program.

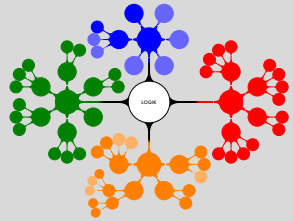


The OpenMP Programming Model VIII

Using the *parallel directive*;

```
1 pragma omp parallel if (is_parallel == 1) num_threads(8)
      private (a) shared (b) firstprivate(c)
2 {
3     /* structured block */
4 }
```

- Here, if the value of the variable *is_parallel* equals one, eight threads are created.
- Each of these threads gets private copies of variables *a* and *c*, and shares a single value of variable *b*.
- Furthermore, the value of each copy of *c* is initialized to the value of *c* before the parallel directive.
- The clause *default (shared)* implies that, by default, a variable is shared by all the threads.
- The clause *default (none)* implies that the state of each variable used in a thread must be explicitly specified.
  - This is generally recommended, to guard against errors arising from unintentional concurrent access to shared data.



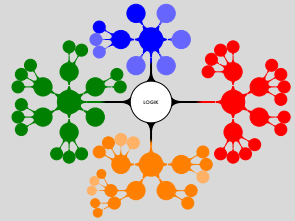
The OpenMP Programming Model IX

The *reduction* clause :

- Specifies how **multiple local copies** of a variable at different threads are **combined into a single copy** at the master when threads exit.
- The usage of the *reduction* clause is *reduction (operator: variable list)*.
  - This clause performs a reduction on the scalar variables specified in the list using the *operator*.
  - The variables in the list are implicitly specified as being private to threads.
- The *operator* can be one of

  `+ * - & | ^ && ||`
- Each of the eight threads gets a copy of the variable *sum*.

```
1 #pragma omp parallel reduction(+: sum) num_threads(8)
2 {
3   /* compute local sums here */
4 }
5 /* sum here contains sum of all local instances of sums */
```

The OpenMP Programming Model X

Parallel Loop:

- Compiler calculates **loop bounds** for each thread directly from serial source (computation decomposition).
- Compiler also manages data partitioning.
- Synchronization also automatic (barrier).
- Preprocessor calculates loop bounds and divide iterations among parallel threads.

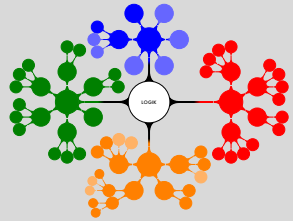
Serial Program:

```
void main()
{
   double Res[1000];

   for(int i=0;i<1000;i++) {
         do_huge_comp(Res[i]);
   }
}
```

Parallel Program:

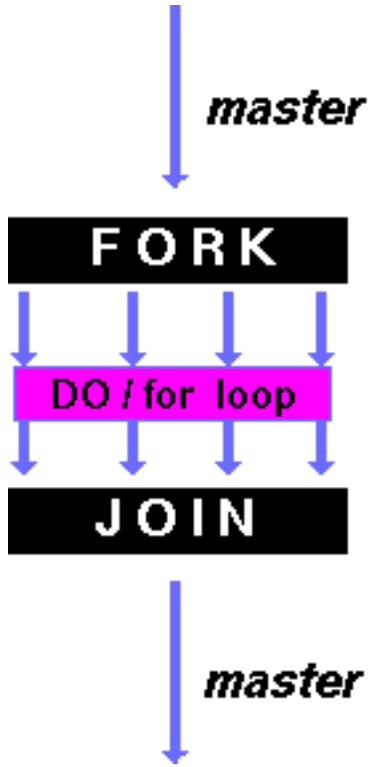
```
void main()
{
   double Res[1000];
#pragma omp parallel for
   for(int i=0;i<1000;i++) {
         do_huge_comp(Res[i]);
   }
}
```

The OpenMP Programming Model XI

Loop Scheduling in Parallel *for* **pragma**

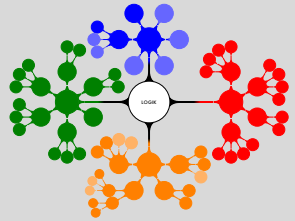
- Master thread creates additional threads, each with a separate execution context.
- All variables declared outside for loop are shared by default, except for loop index which is private per thread.
- Implicit "barrier" synchronization at end of for loop.
- Divide index regions sequentially per thread
  - Thread 0 gets $0, 1, \ldots (max/n) - 1$
  - Thread 1 gets $max/n, max/n + 1, \ldots 2 * (max/n) - 1$
  - ⋮



Example:

```
#pragma omp parallel for
for (i=0; i<max; i++)
     zero[i] = 0;
```

- Breaks for loop into chunks, and allocate each to a separate thread.
- if max = 1000 with 2 threads: assign 0-499 to thread 0, and 500-999 to thread 1.



The OpenMP Design Concepts

- Load balance, Scheduling overhead, Data locality, Data sharing, Synchronization.
- OpenMP is a compiler-based technique to create concurrent code from (mostly) serial code.
- OpenMP can enable (easy) parallelization of loop-based code with fork-join parallelism.

```
1 pragma omp parallel
2 pragma omp parallel for
3 pragma omp parallel private ( i, x )
4 pragma omp atomic
5 pragma omp critical
6 pragma omp for reduction(+ : sum)
```

- OpenMP performs comparably to manually-coded threading.