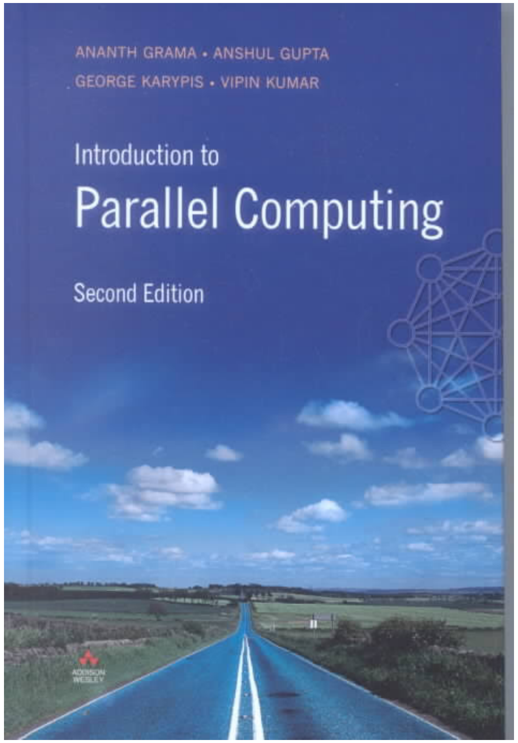
Programming Shared-Memory Platforms with OpenMP



Most slides taken from Chapter 7 of **Introduction to Parallel Computing** by Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar

OpenMP: A Standard for Directive Based Parallel Programming

https://www.openmp.org

OpenMP:

- One of the most common parallel programming models in use today.
- Relatively easy to use, which makes it a great language to start with when learning how to write parallel software.
- Standardizes last 20+ years of SMP practice.
- Has seen various revisions/extensions
  - Most recent: 5.0 (November 2018).

OpenMP: An API for Writing Multithreaded Applications

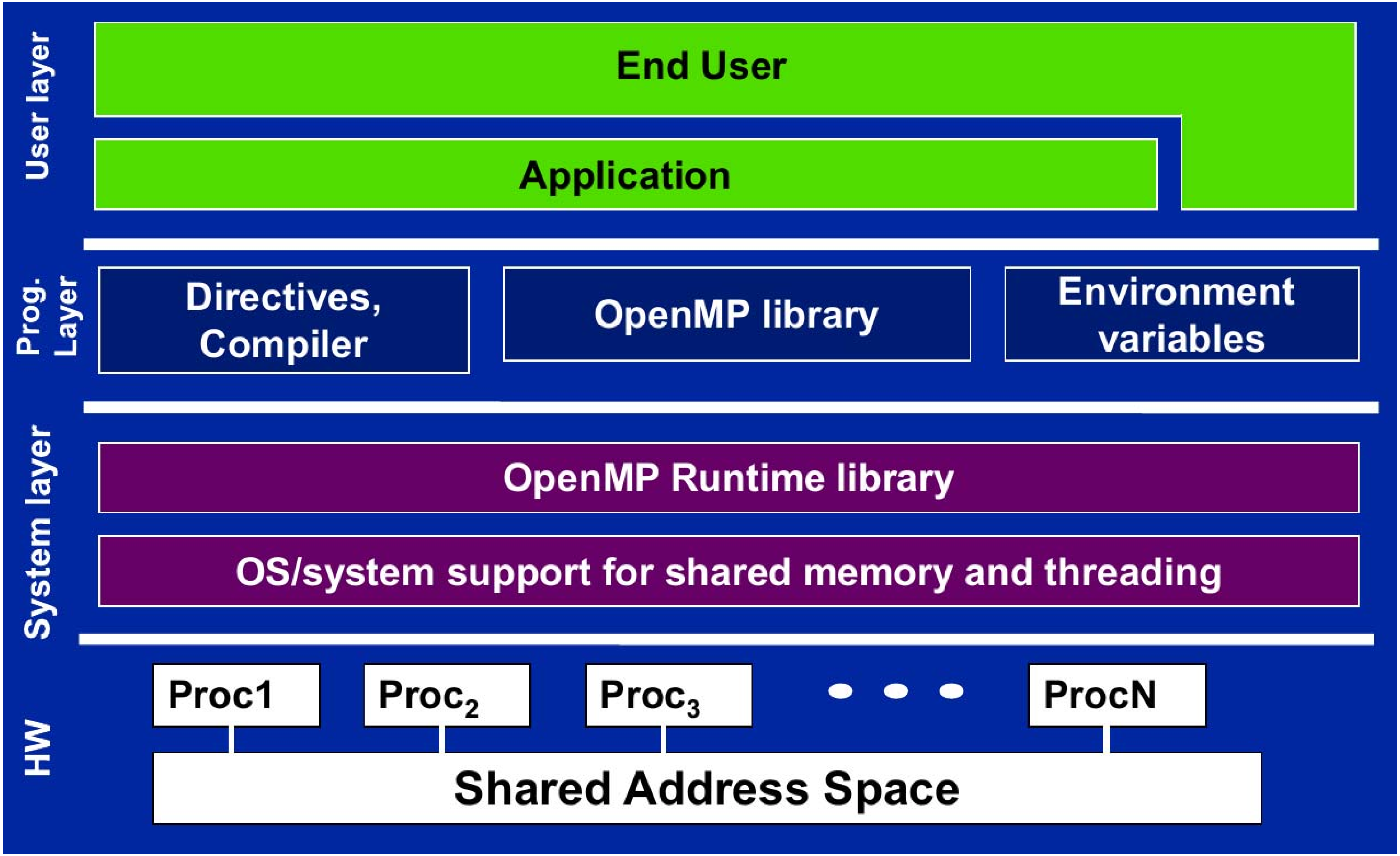
OpenMP:

- A set of compiler directives and library routines for parallel application programmers
- Greatly simplifies writing multi-threaded shared memory programs in FORTRAN, C and C++.

OpenMP directives:

- Provide support for concurrency, synchronization, and data handling.
- Obviate the need for explicitly setting up mutexes, condition variables, data scope, and initialization.

OpenMP Solution Stack



OpenMP Programming Model

- OpenMP directives in C and C++ are based on the **`#pragma`** compiler directives.
- A directive consists of a directive name followed by a list of clauses.

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

  - Example:

```
#pragma omp parallel num_threads(8)
```

- Function prototypes and types in the file:

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

OpenMP Programming Model

- OpenMP programs execute serially until they encounter the **`parallel`** directive, which creates a group of threads.

```
#pragma omp parallel
/* structured block */
```

- Most OpenMP constructs apply to a “structured block.”
  - **Structured block**: A block with one or more statements with one point of entry at the top and one point of exit at the bottom.
  - It’s OK to have an **`exit()`** within the structured block.
- The main thread that encounters the **`parallel`** directive becomes the *master* of this group of threads and is assigned the thread id 0 within the group.

Hello OpenMP World!

Let's write a hello OpenMP world program.

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

int main() {
  #pragma omp parallel
  {
    int id = omp_get_thread_num();

    printf(" hello(%d)", id);
    printf(" world(%d)", id);
  }
  printf("\n");
  return 0;
}
```

OpenMP include file

Parallel region with default number of threads

Runtime library function to return a thread ID

End of parallel region

for the bash shell

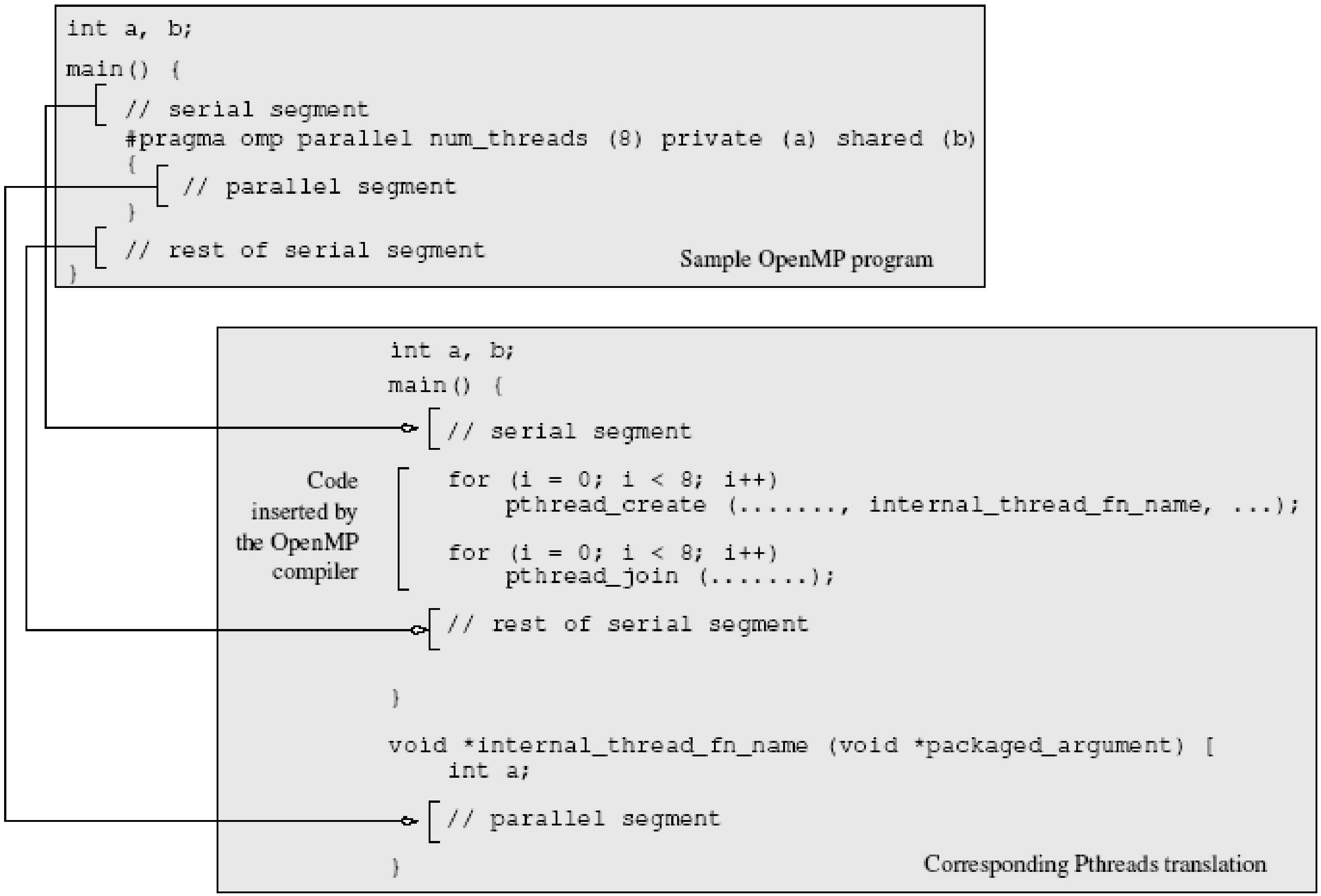
```
> gcc -fopenmp hello.c
> export OMP_NUM_THREADS=8
> ./a.out
```

OpenMP Programming Model

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

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

OpenMP Programming Model



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

OpenMP Programming Model

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

- 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`**.
- The value of each copy of **`c`** is initialized to the value of **`c`** before the parallel directive.
- The default state of a variable is specified by the clause default (shared) or default (none).

Reduction Clause in OpenMP

- 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)`**.
- The variables in the list are implicitly specified as being private to threads.
- The **`operator`** can be one of **+**, *****, **-**, **&**, **|**, **^**, **&&**, and **||**.

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

OpenMP Programming: Example

```
/* ******************************************************
/* An OpenMP version of a threaded program to compute PI.
/* ******************************************************/
#pragma omp parallel default(private) shared (npoints) \
    reduction(+: sum) num_threads(8)
{
  num_threads = omp_get_num_threads();
  sample_points_per_thread = npoints / num_threads;
  sum = 0;
  for (i = 0; i < sample_points_per_thread; i++) {
    rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
    rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
    if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
         (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
      sum ++;
  }
}
```

Specifying Concurrent Tasks in OpenMP

- The **`parallel`** directive can be used in conjunction with other directives to specify concurrency across iterations and tasks.
- OpenMP provides two directives - **`for`** and **`sections`** - to specify concurrent iterations and tasks.
- The **`for`** directive is used to split parallel iteration spaces across threads. The general form of a **`for`** directive is:

```
#pragma omp for [clause list]
    /* for loop */
```

- The clauses that can be used in this context are: **`private`**, **`firstprivate`**, **`lastprivate`**, **`reduction`**, **`schedule`**, **`nowait`**, and **`ordered`**.

Specifying Concurrent Tasks in OpenMP: Example

```
#pragma omp parallel default(private) shared (npoints) \
    reduction(+: sum) num_threads(8)
{
  sum = 0;
  #pragma omp for
  for (i = 0; i < npoints; i++) {
    rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
    rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
    if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
         (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
      sum ++;
  }
}
```

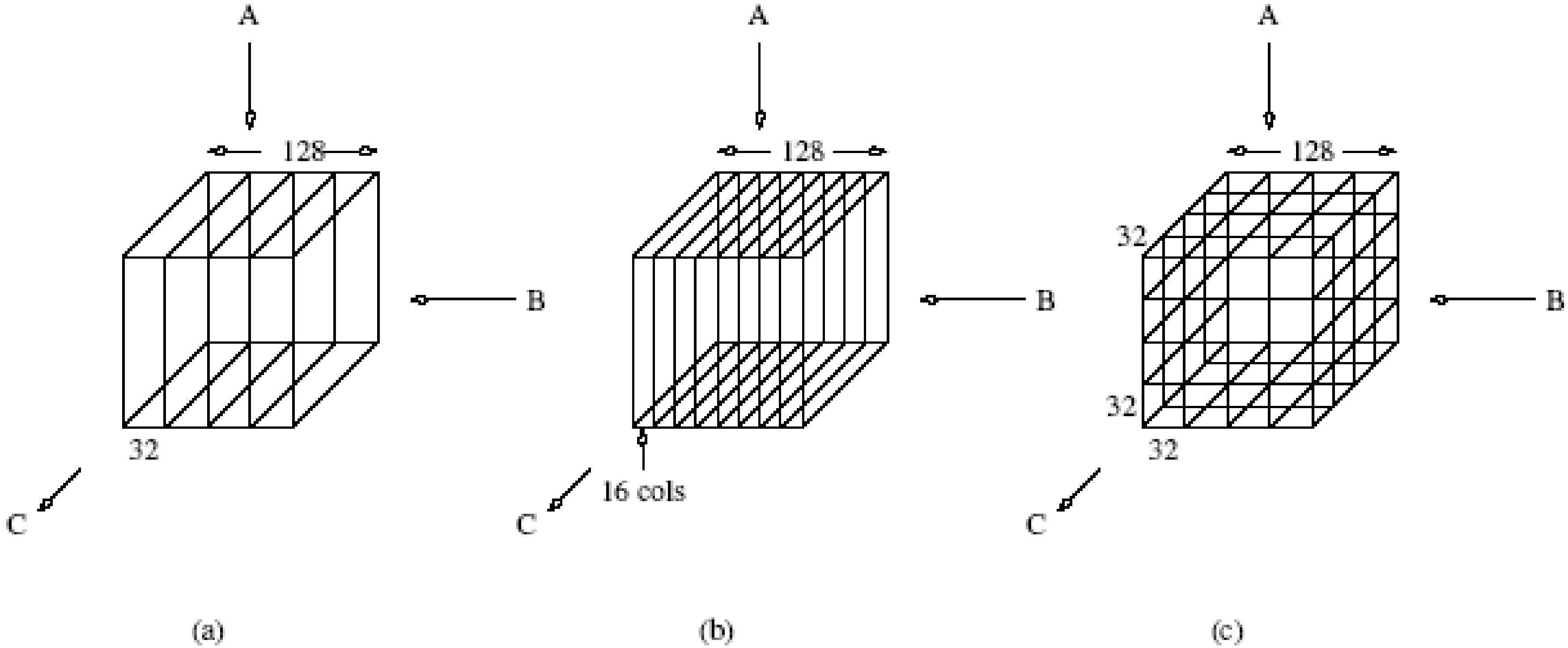
Assigning Iterations to Threads

- The **`schedule`** clause of the **`for`** directive deals with the assignment of iterations to threads.
- The general form of the **`schedule`** clause is **`schedule(scheduling_class[, parameter])`**.
- OpenMP supports four scheduling classes: **`static`**, **`dynamic`**, **`guided`**, and **`runtime`**.

Assigning Iterations to Threads: Example

```
/* static scheduling of matrix multiplication loops */
#pragma omp parallel default(private) shared (a, b, c, dim) \
    num_threads(4)
    #pragma omp for schedule(static)
    for (i = 0; i < dim; i++) {
        for (j = 0; j < dim; j++) {
            c(i,j) = 0;
            for (k = 0; k < dim; k++) {
                c(i,j) += a(i, k) * b(k, j);
            }
        }
    }
```

Assigning Iterations to Threads: Example



Three different schedules using the static scheduling class of OpenMP.

Parallel For Loops

- Often, it is desirable to have a sequence of **`for`** directives within a parallel construct that do not execute an implicit barrier at the end of each **`for`** directive.
- OpenMP provides a clause - **`nowait`**, which can be used with a **`for`** directive.

Parallel For Loops: Example

```
#pragma omp parallel
{
  #pragma omp for nowait
    for (i = 0; i < nmax; i++)
      if (isEqual(name, current_list[i])
        processCurrentName(name);
  #pragma omp for
    for (i = 0; i < mmax; i++)
      if (isEqual(name, past_list[i])
        processPastName(name);
}
```

The **sections** Directive

- OpenMP supports non-iterative parallel task assignment using the **sections** directive.
- The general form of the **sections** directive is:

```
#pragma omp sections [clause list]
{
   [#pragma omp section
      /* structured block */
   ]
   [#pragma omp section
      /* structured block */
   ]
   ...
}
```

The `sections` Directive: Example

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            taskA();
        }
        #pragma omp section
        {
            taskB();
        }
        #pragma omp section
        {
            taskC();
        }
    }
}
```

Nesting `parallel` Directives

- Nested parallelism can be enabled using the **`OMP_NESTED`** environment variable.
- If the **`OMP_NESTED`** environment variable is set to **`TRUE`**, nested parallelism is enabled.
- In this case, each parallel directive creates a new team of threads.

Synchronization Constructs

OpenMP provides a variety of synchronization constructs:

```
#pragma omp barrier
#pragma omp single [clause list]
   /* structured block */
#pragma omp master
   /* structured block */
#pragma omp critical [(name)]
   /* structured block */
#pragma omp ordered
   /* structured block */
```

OpenMP Library Functions

In addition to directives, OpenMP also supports a number of functions that allow a programmer to control the execution of threaded programs.

```
/* thread and processor count */
void omp_set_num_threads (int num_threads);
int omp_get_num_threads ();
int omp_get_max_threads ();
int omp_get_thread_num ();
int omp_get_num_procs ();
int omp_in_parallel ();
```

OpenMP Library Functions

```
/* controlling and monitoring thread creation */
void omp_set_dynamic (int dynamic_threads);
int omp_get_dynamic ();
void omp_set_nested (int nested);
int omp_get_nested ();

/* mutual exclusion */
void omp_init_lock (omp_lock_t *lock);
void omp_destroy_lock (omp_lock_t *lock);
void omp_set_lock (omp_lock_t *lock);
void omp_unset_lock (omp_lock_t *lock);
int omp_test_lock (omp_lock_t *lock);
```

In addition, all lock routines also have a nested lock counterpart for recursive mutexes.

Environment Variables in OpenMP

OMP_NUM_THREADS: Specifies the default number of threads created upon entering a parallel region.

OMP_SET_DYNAMIC: Determines if the number of threads can be dynamically changed.

OMP_NESTED: Turns on nested parallelism.

OMP_SCHEDULE: Scheduling of for-loops if the clause specifies runtime.

Explicit Threads versus Directive Based Programming

- Directives layered on top of threads facilitate a variety of thread-related tasks.
- A programmer is rid of the tasks of initializing attributes objects, setting up arguments to threads, partitioning iteration spaces, etc.

Explicit Threads versus Directive Based Programming

There are some drawbacks to using directives as well.

- With explicit threading data exchange is more apparent. This helps in alleviating some of the overheads from data movement, false sharing, and contention.
- Explicit threading also provides a richer API in the form of condition waits, locks of different types, and increased flexibility for building composite synchronization operations.
- Since explicit threading is used more widely than OpenMP, tools and support for Pthreads programs are easier to find.