Lab # 4

- **OpenMP Written Exam in first 20 minutes of Lab # 5**
- OpenMP Solution stack and Parallel execution model
- Sections
- Data handing in OpenMP

Nesting `parallel` Directives (covered in Lab #3)

```
#pragma omp parallel for default(private) shared (a, b, c, dim) \
                    num_threads(2)
    for (i = 0; i < dim; i++) {
    #pragma omp parallel for default(private) shared (a, b, c, dim) \
                    num_threads(2)
        for (j = 0; j < dim; j++) {
            c(i,j) = 0;
            #pragma omp parallel for default(private) \
                    shared (a, b, c, dim) num_threads(2)
            for (k = 0; k < dim; k++) {
                c(i,j) += a(i, k) * b(k, j);
            }
        }
    }
```

Nesting `parallel` Directives

OMP_NESTED environment variable

set to (TRUE or FALSE)

We start by making a few observations about how this segment is written. Instead of nesting three `for` directives inside a single `parallel` directive, we have used three `parallel for` directives. This is because OpenMP does not allow `for`, `sections`, and `single` directives that bind to the same `parallel` directive to be nested. Furthermore, the code as written only generates a logical team of threads on encountering a nested `parallel` directive. The newly generated logical team is still executed by the same thread corresponding to the outer `parallel` directive. To generate a new set of threads, nested parallelism must be enabled using the `OMP_NESTED` environment variable. If the `OMP_NESTED` environment variable is set to `FALSE`, then the inner `parallel` region is serialized and executed by a single thread. If the `OMP_NESTED` environment variable is set to `TRUE`, nested parallelism is enabled. The default state of this environment variable is `FALSE`, i.e., nested parallelism is disabled. OpenMP environment

OpenMP environment variables

- OpenMP environment variables are used to control the behavior of OpenMP programs at runtime by setting specific parameters and options.
- These variables are set externally in the shell environment.

✓ OMP_NUM_THREADS
✓ OMP_NESTED
✓ OMP_SCHEDULE (for default schedule clause)
✓ OMP_THREAD_LIMIT

Total # of threads upper limit.

```
setenv OMP_SCHEDULE "static,4"
```

```
//Linux-Unix
export OMP_NUM_THREADS=4
./my_openmp_program

// windows
set OMP_NUM_THREADS=4
my_openmp_program.exe

//include bash script
#!/bin/bash
export OMP_NUM_THREADS=4
./my_openmp_program
```

The `sections` Directive

The `for` directive is suited to partitioning iteration spaces across threads. Consider now a scenario in which there are three tasks (`taskA`, `taskB`, and `taskC`) that need to be executed. Assume that these tasks are independent of each other and therefore can be assigned to different threads. OpenMP supports such non-iterative parallel task assignment using the `sections` directive. The general form of the `sections` directive is as follows:

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

This `sections` directive assigns the structured block corresponding to each section to one thread (indeed more than one section can be assigned to a single thread). The `clause list` may include the following clauses – `private`, `firstprivate`, `lastprivate`, `reduction`, and `nowait`. The syntax and semantics of these clauses are identical to those in the case of the `for` directive. The `lastprivate` clause, in this case, specifies that the last section (lexically) of the `sections` directive updates the value of the variable. The `nowait` clause specifies that there is no implicit synchronization among all threads at the end of the `sections` directive.

Sections

- Sections will be executed in parallel
- Can combine parallel and section like we did with for
- If more threads than sections then idle threads will exist
- If less threads than sections then some sections will execute in serial

For executing the three concurrent tasks `taskA`, `taskB`, and `taskC`, the corresponding `sections` directive is as follows:

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

Merging Directives

```
#pragma omp parallel sections
{
    #pragma omp section
    {
        taskA();
    }
    #pragma omp section
    {
        taskB();
    }
    /* other sections here */
}
```

Sections

```
#define N 1000
main (){
  int i;float a[N], b[N], c[N];
  for (i=0; i < N; i++) a[i] = b[i] = … ;

# pragma omp parallel shared(a,b,c) private(i)
  {
   . . .
# pragma omp sections
  {
#   pragma omp section
    {
      for (i=0; i < N/2; i++) c[i] = a[i] + b[i];
    }
#   pragma omp section
    {
      for (i=N/2; i < N; i++) c[i] = a[i] + b[i];
    }
  } /* end of sections */
. . .
 } /* end of parallel */
}
```

Synchronization Point: The `barrier` Directive

A barrier is one of the most frequently used synchronization primitives. OpenMP provides a `barrier` directive, whose syntax is as follows:

```
1        #pragma omp barrier
```

- **When a thread executes a barrier it will wait until all other threads in the team also execute the barrier.**
- **Then threads continue working as usual.**

On encountering this directive, all threads in a team wait until others have caught up, and then release. When used with nested `parallel` directives, the `barrier` directive binds to the closest `parallel` directive. For executing barriers conditionally, it is important to note that a `barrier` directive must be enclosed in a compound statement that is conditionally executed. This is because pragmas are compiler directives and not a part of the language. Barriers can also be effected by ending and restarting `parallel` regions. However, there is usually a higher overhead associated with this. Consequently, it is not the method of choice for implementing barriers.

```c
int main() {
    int num_threads = 4;

    #pragma omp parallel num_threads(num_threads)
    {
        int thread_id = omp_get_thread_num();
        printf("Thread %d is doing some work.\n", thread_id);

        // Barrier synchronization point
        #pragma omp barrier

        printf("Thread %d finished its work after the barrier.\n", thread_id);
    }

    return 0;
}
```

Single Thread Executions: The `single` and `master` Directives

A `single` directive specifies a structured block that is executed by a single (arbitrary) thread. The syntax of the `single` directive is as follows:

```
1    #pragma omp single [clause list]
2             structured block
```

The clause list can take clauses `private`, `firstprivate`, and `nowait`. These clauses have the same semantics as before. On encountering the `single` block, the first thread enters the block. All the other threads proceed to the end of the block. If the `nowait` clause has been specified at the end of the block, then the other threads proceed; otherwise they wait at the end of the `single` block for the thread to finish executing the block. This directive is useful for computing global data as well as performing I/O.

Single Thread Executions: The `single` and `master` Directives

The `master` directive is a specialization of the `single` directive in which only the master thread executes the structured block. The syntax of the `master` directive is as follows:

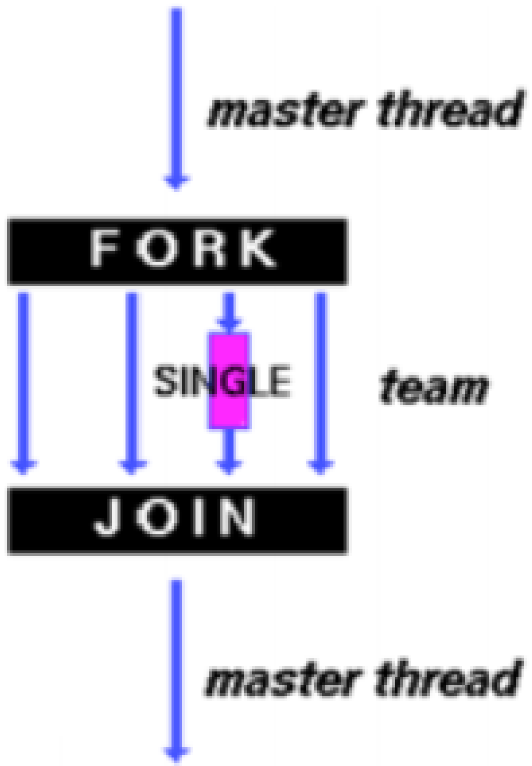
```
#pragma omp master
          structured block
```

In contrast to the `single` directive, there is no implicit barrier associated with the `master` directive.

Single and Master

- **Single indicates that only one thread in team will execute the code.**

  #pragma omp single



- **Master indicates that only the master thread will execute the code.**

  #pragma omp master

```c
int main() {
    int shared_data = 0;

    #pragma omp parallel
    {
        // This block of code will be executed by all threads
        int thread_id = omp_get_thread_num();
        printf("Thread %d: Executing parallel section.\n", thread_id);

        // Use #pragma omp single to execute code only once by one thread
        #pragma omp single
        {
            shared_data = 42;
            printf("Thread %d: Setting shared_data to %d.\n", thread_id, shared_data);
        }
    }

    // After the parallel region, all threads have the same value of shared_data
    printf("Value of shared_data after parallel region: %d\n", shared_data);

    return 0;
}
```

```c
int main() {
    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num();
        int num_threads = omp_get_num_threads();

        #pragma omp master
        {
            // This block of code is executed only by the master thread
            printf("I am the master thread (Thread %d out of %d)\n", thread_id, num_threads);
        }

        // All threads participate in this work
        printf("Thread %d doing some work\n", thread_id);
    }

    return 0;
}
```

Critical with Names

- **Critical ensures that only one thread can execute code block at a time.**

```
# pragma omp critical
  global_result += my_result ;
```

- **You can name critical sections, then critical sections with different names can be executed in parallel**

```
# pragma omp critical(name)
  global_result += my_result ;
```

In OpenMP, when you have multiple #pragma omp critical statements with the same name in a parallel region, it allows different sections of your code to be protected by separate critical sections, but with the same name acting as a unique identifier. Each critical section with the same name will ensure that only one thread can execute the code within that specific critical section at a time, even if the sections are in different parts of the parallel region.

Example 7.16 Using the critical directive for producer-consumer threads

```
#pragma omp parallel sections
{
    #pragma parallel section
    {
        /* producer thread */
        task = produce_task();
        #pragma omp critical ( task_queue)
        {
            insert_into_queue(task);
        }
    }
    #pragma parallel section
    {
        /* consumer thread */
        #pragma omp critical ( task_queue)
        {
            task = extract_from_queue(task);
        }
        consume_task(task);
    }
}
```

- A producer thread generates a task and inserts it into a task-queue.
- The consumer thread extracts tasks from the queue
  and executes them one at a time.
- Since there is concurrent access to the task-queue, these accesses must be serialized using critical blocks.
- Specifically, the tasks of inserting and extracting from the task-queue must be serialized.
- Note that queue full and queue empty conditions must be explicitly handled here in functions insert_into_queue and extract_from_queue.

7.10.4 Data Handling in OpenMP

Scope

- In serial programming, the scope of a variable consists of those parts of a program in which the variable can be used.
- In OpenMP, the scope of a variable refers to the set of threads that can access the variable in a parallel block.

Scope in OpenMP

- A variable that can be accessed by all the threads in the team has shared scope.
- A variable that can only be accessed by a single thread has private scope.
- The default scope for variables declared before a parallel block is shared.

7.10.4 Data Handling in OpenMP

One of the critical factors influencing program performance is the manipulation of data by threads. We have briefly discussed OpenMP support for various data classes such as `private`, `shared`, `firstprivate`, and `lastprivate`. We now examine these in greater detail, with a view to understanding how these classes should be used. We identify the following heuristics to guide the process:

- If a thread initializes and uses a variable (such as loop indices) and no other thread accesses the data, then a local copy of the variable should be made for the thread. Such data should be specified as `private`.
- If a thread repeatedly reads a variable that has been initialized earlier in the program, it is beneficial to make a copy of the variable and inherit the value at the time of thread creation. This way, when a thread is scheduled on the processor, the data can reside at the same processor (in its cache if possible) and accesses will not result in interprocessor communication. Such data should be specified as `firstprivate`.
- If multiple threads manipulate a single piece of data, one must explore ways of breaking these manipulations into local operations followed by a single global operation. For example, if multiple threads keep a count of a certain event, it is beneficial to keep local counts and to subsequently accrue it using a single summation at the end of the parallel block. Such operations are supported by the `reduction` clause.
- If multiple threads manipulate different parts of a large data structure, the programmer should explore ways of breaking it into smaller data structures and making them private to the thread manipulating them.

Data scope attribute clause	Description
private	The **private** clause declares the variables in the list to be private to each thread in a team.
firstprivate	The **firstprivate** clause provides a superset of the functionality provided by the private clause. The private variable is initialized by the original value of the variable when the parallel construct is encountered.
lastprivate	The **lastprivate** clause provides a superset of the functionality provided by the private clause. The private variable is updated after the end of the parallel construct.
shared	The **shared** clause declares the variables in the list to be shared among all the threads in a team. All threads within a team access the same storage area for shared variables.
reduction	The **reduction** clause performs a reduction on the scalar variables that appear in the list, with a specified operator.
default	The **default** clause allows the user to affect the data-sharing attribute of the variables appeared in the parallel construct.

Scoping in OpenMP: Dividing variables in *shared* and *private*:

→ *private*-list and *shared*-list on Parallel Region

→ *private*-list and *shared*-list on Worksharing constructs

→ General default is *shared* for Parallel Region.

→ Loop control variables on *for*-constructs are *private*

→ Non-static variables local to Parallel Regions are *private*

→ *private*: A new uninitialized instance is created for each thread

  → *firstprivate*: Initialization with Master's value

  → *lastprivate*: Value of last loop iteration is written back to Master

→ Static variables are *shared*

The default clause

- Lets the programmer specify the scope of each variable in a block.

```
default(none)
```

- With this clause the compiler will require that we specify the scope of each variable we use in the block and that has been declared outside the block.

7.10.5 OpenMP Library Functions

Controlling Number of Threads and Processors

```
#include <omp.h>

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();
```

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

int main() {
    int num_processors = omp_get_num_procs(); // Get the number of available processors
    printf("Available processors: %d\n", num_processors);

    int max_threads = num_processors; // Set the maximum number of threads to the number of processors
    omp_set_num_threads(max_threads);

    // Now, create a parallel region and check the number of threads
    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num();
        int num_threads = omp_get_num_threads();

        printf("Thread %d out of %d threads.\n", thread_id, num_threads);
    }

    return 0;
}
```