French-Azerbaijani University

Parallel Programming

Practical Session 4 : OpenMP

Lecturers and Instructors :

Eldar Babayev — Salah El Falou — Rabih Amhaz

December 20, 2020

1 Exercise: Number of Threads

1.1 Corrected Code

```
// Exercise 1 - Corrected Parallelized Version
#include <stdio.h>
#include <omp.h>
int main() {
    size_t nb_threads = 0;
    #pragma omp parallel
    {
        #pragma omp critical //or atomic
        nb_threads++;
    }
    printf("nb_threads = %zu\n", nb_threads);
    return 0;
}
```

Listing 1: Finding Number of Threads in Parallel

So, to handle the counting process of number of threads in a parallelized way, we could use of a `critical` or `atomic` section, if we are to avoid the `reduction`, of course.

- The original code lacked the usage of critical (or atomic) section, and the reason we need such sections is that each thread would increment the `nb_threads` variable independently thence the result would be unrelated to the actual number of threads of the machine.
- Critical directive allows the block of code within it to be run by one thread a time.
- Whereas atomic specifies that a memory location will be updated atomically.

2 Exercise: First Prime Numbers

2.1 Parallelized Code

```c
// Exercise 2 - Parallelized Version
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <omp.h>
#define PRIME_MIN 3
#define PRIME_MAX 101

int main() {
    size_t primes[PRIME_MAX];
    size_t nb_primes = 0;
    size_t i, divisor;
    bool is_prime;
    #pragma omp parallel private(divisor,is_prime)
    {
        #pragma omp for ordered
        for (i = PRIME_MIN; i < PRIME_MAX; i += 2) {
            #pragma omp ordered
            {
                is_prime = true;
                divisor = PRIME_MIN;
                while((divisor < i) && is_prime) {
                    if ((i % divisor) == 0) { is_prime = false; }
                    divisor += 2;
                }
                if (is_prime) {
                    #pragma omp critical
                    {
                        primes[nb_primes] = i;
                        nb_primes++;
                    }
                }
            }
        }
    }
    printf("\nNumber of Primes = %zu ~ (Between %d and %d)\n",nb_primes,PRIME_MIN,
   PRIME_MAX);
    for ( int a = 0; a < nb_primes; a++ ) {
        printf("%zu, ",primes[a]);
    }
    puts("\n");
    return 0;
}
```

Listing 2: First Prime Numbers Parallelized

2.2 Output

```
faxmishok@Lenovo-V110:~/Desktop/PP_PW4_Fahmin_Guliyev$ gcc ex2.c -o ex2 -fopenmp
faxmishok@Lenovo-V110:~/Desktop/PP_PW4_Fahmin_Guliyev$ ./ex2

Number of Primes = 24 ~ (Between 3 and 101)
3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79,

faxmishok@Lenovo-V110:~/Desktop/PP_PW4_Fahmin_Guliyev$
```

As can be seen from above figures, the program is parallelized and works just fine with 4 threads.

- The implementation consists of a parallel region (as usual), a private clause for some variables, an ordered for region and a critical section for careful incrementation.
- Our `divisor` and `is_prime` variables are inside set inside `private`, because each thread should have its own copy of these variables, otherwise the loop will be mess.
- Ordered for directive specifies that code under a parallelized for loop should be executed like a sequential loop, which allows us to fill all primes respectively into the `primes` array with order. If it's not used then the primes will be unordered in the array.
- The main point here, is to move the **2 instructions** which are essential for the whole task inside a critical region, because it will work correctly only a single thread runs those intructions. They are 1) filling the array and 2) incrementing the prime number count.

3 Exercise: Synchronization Using Lock

3.1 Code

```c
// Exercise 3 - Corrected Parallelized Version with Locks
#include <omp.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

int main() {
    int p;
    omp_lock_t lock1, lock2;

    omp_init_lock(&lock1);
    omp_init_lock(&lock2);
    omp_set_lock(&lock1);
    omp_set_lock(&lock2);
    #pragma omp parallel sections private(p) default(shared)
    {
        #pragma omp section
        {
            p = omp_get_thread_num();
            printf("Th%d: Hello\n",p);
            omp_unset_lock(&lock1);
        }
        #pragma omp section
        {
            omp_set_lock(&lock1);
            p = omp_get_thread_num();
            printf("Th%d: World\n",p);
            omp_unset_lock(&lock1);
            omp_unset_lock(&lock2);

        }
        #pragma omp section
        {
            omp_set_lock(&lock2);
            p = omp_get_thread_num();
            printf("Th%d: Bye\n",p);
            omp_unset_lock(&lock2);
        }
    }
    omp_destroy_lock(&lock1);
    omp_destroy_lock(&lock2);
    return 0;
}
```

Listing 3: Synchronization Using Locks

The OpenMP runtime library includes a set of general-purpose lock routines that can be used for synchronization. These general-purpose lock routines operate on OpenMP locks that are represented by OpenMP lock variables.

Locks simply allow a particular part of the code run by one thread, and no other thread can interrupt or bother that thread or task until it finishes.

- As expected, I decline 2 lock variables of type `omp_lock_t`.
- Then I initialize these locks (allocate).
- After initialization, right before entering the parallel region, I set both locks by `omp_set_lock()` function. The reason I do it because when setting locks inside a parallel section, it will serve only for the current section or thread. To handle it in a general way, I must set locks before entering the parallel region.
- I set first lock first, then the second, then here comes the parallel region with sections.
- Variable `p`, which will represent the thread id or number has to be enclosed with private directive to make sure each thread will have its own `p`.
- In the first section, I leave the 2 instructions as the same as in the original code, then I unset (or unlock) the first lock. I did not do any operation regarding to the locks in the first parallel section because I want the word *Hello* to be printed first. So both locks are set, nobody can bother this thread.
- Second parallel section depends on first lock, so before exiting the first parallel section, I unset the first lock, and finally it can enter to the second section.
- But why would it enter directly to the second section? It could enter the third, but third parallel section has its own requirements too, as we can see both locks must be unset to enter the third section, so that's why it enters to the second section before it does to third one.
- Just before exiting the second parallel section, we need to unset both locks, to make sure everything is released, so that the condition is fulfilled to complete and enter to the third section of code.
- At the end the locks are destroyed to free excess memory.