CSE 4001: Lab Assessment #2

Due on Monday, Aug 6, 2018

Prof. Deebak BD 5:30pm

Jacob John

Contents

Problem 1

Write a simple OpenMP program to define a parallel region. It should be helpful to understand the purposes of shared variable and local or private variable to be executed in each thread in OpenMP .

Note: Note: A prefix of *#pragma omp* is used as an OpenMP directive construct to exploit the feature of concurrency.

Explanation

The following code is implemented in OpenMP for Shared Memory Parallelization specifications. The OpenMP API specification provides offers portability across shared memory architectures for parallel programming.

In the first program, *shared(a)* is implemented where a is one more more variables to share by multiple threads. All variables declared outside the parallel block can be accessed by the threads if they are declared as shared.

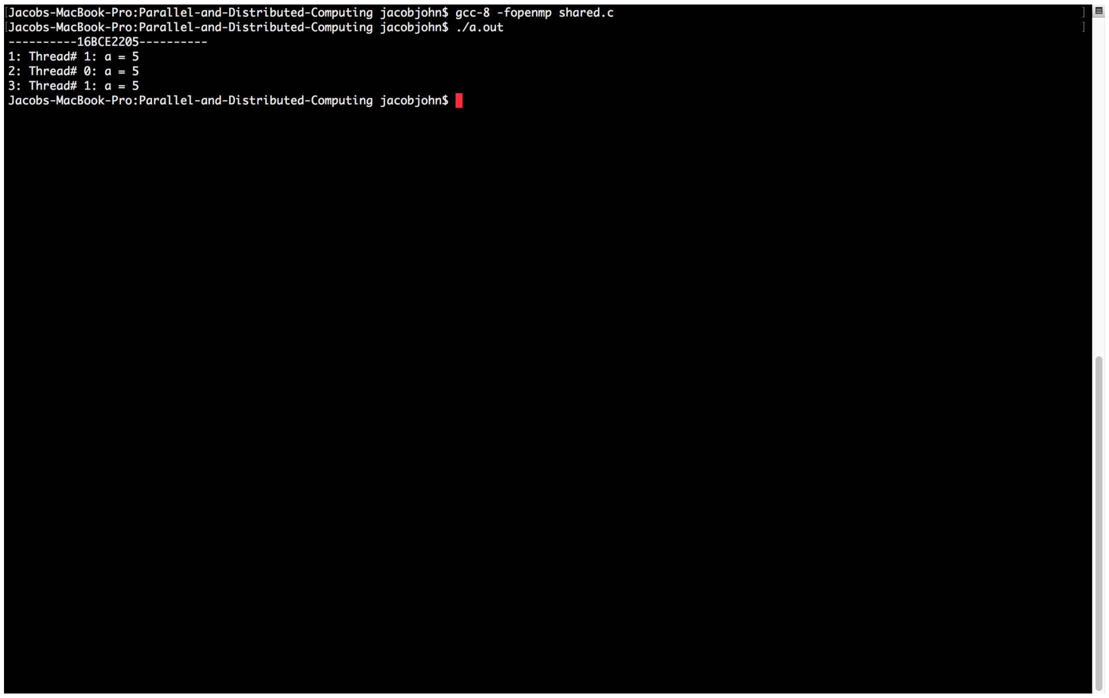
The following program utilizes a race condition, thereby, the value of a could be either 2 or 5, depending on the timing of the threads, and the implementation of the assignment to a. The barrier after the first print statement contains implicit flushes on all threads, as well as a thread synchronization. In this case a print of 5 will be guaranteed by 2 and 3.

Listing 1: Shared variable using OpenMP in C

```c
/*Shared variables in openMP*/
#include <stdio.h>
#include <omp.h>
int main()
{
    int a = 2;
    printf("----------16BCE2205----------\n");
#pragma omp parallel num_threads(2) shared(a)
    {
        if (omp_get_thread_num() == 0)
            a = 5;
        else
            printf("1: Thread# %d: a = %d\n", omp_get_thread_num(), a);
            /* Print 1: the following read of x has a race */
#pragma omp barrier
        if (omp_get_thread_num() == 0)
            printf("2: Thread# %d: a = %d\n", omp_get_thread_num(), a);
            /* Print 2 */
        else
            printf("3: Thread# %d: a = %d\n", omp_get_thread_num(), a);
            /* Print 3 */
    }
    return 0;
}
```

Problem 1 continued on next page...

Output:



```
Jacobs-MacBook-Pro:Parallel-and-Distributed-Computing jacobjohn$ gcc-8 -fopenmp shared.c
Jacobs-MacBook-Pro:Parallel-and-Distributed-Computing jacobjohn$ ./a.out
----------16BCE2205----------
1: Thread# 1: a = 5
2: Thread# 0: a = 5
3: Thread# 1: a = 5
Jacobs-MacBook-Pro:Parallel-and-Distributed-Computing jacobjohn$
```

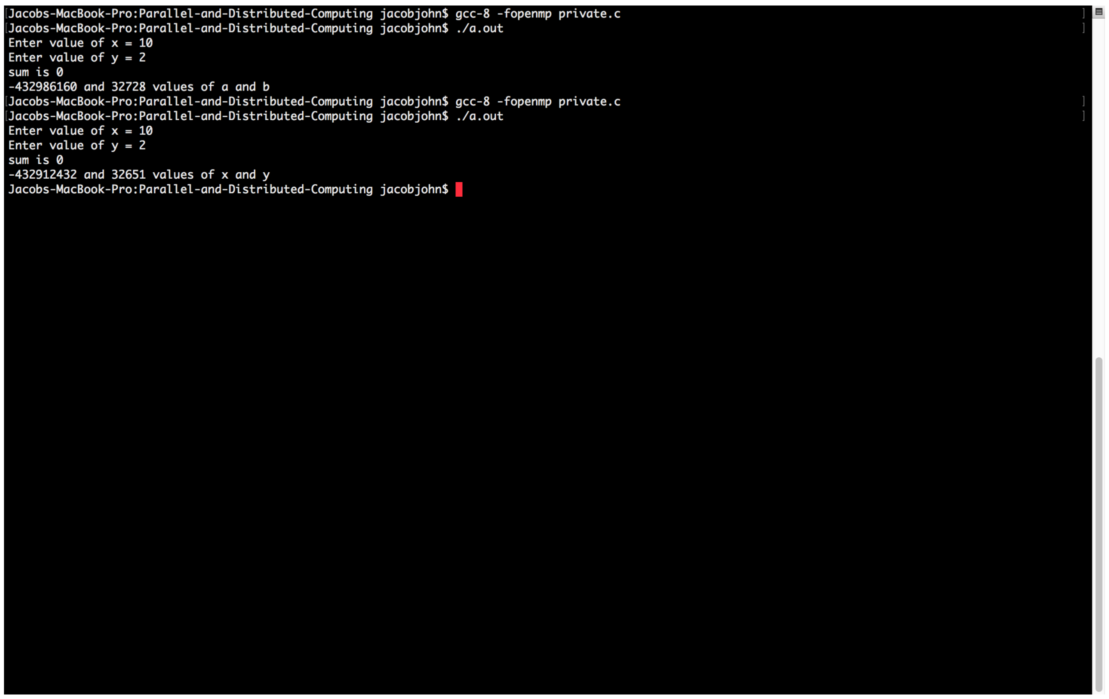
We use *private(var)* when the var variable has to have instances in each thread. Variables initialized in the parallel region are by default private and cannot be accessed by outside code during parallel execution.

As shown in the output below, both the variables produce garbage value when accessed by the second thread. This is because a separate instance in created. The same goes for the first thread as well, since x and y contain garbage values, it produces an output of 0.

Listing 2: Private variable using OpenMP in C

```c
/*Private Variables in OpenMP*/
#include <stdio.h>
#include <omp.h>
int main()
{
    int x, y, tid;
    printf("Enter value of x = ");
    scanf("%d", &x);
    printf("Enter value of y = ");
    scanf("%d", &y);
    omp_set_num_threads(2);
#pragma omp parallel private(x, y, tid)
    {
        tid = omp_get_thread_num();
        if (tid == 0)
            printf("sum is %d\n", (x + y));
        if (tid == 1)
            printf("%d and %d values of x and y\n", x, y);
    }
    return 0;
}
```

Output:



```
Jacobs-MacBook-Pro:Parallel-and-Distributed-Computing jacobjohn$ gcc-8 -fopenmp private.c
Jacobs-MacBook-Pro:Parallel-and-Distributed-Computing jacobjohn$ ./a.out
Enter value of x = 10
Enter value of y = 2
sum is 0
-432986160 and 32728 values of a and b
Jacobs-MacBook-Pro:Parallel-and-Distributed-Computing jacobjohn$ gcc-8 -fopenmp private.c
Jacobs-MacBook-Pro:Parallel-and-Distributed-Computing jacobjohn$ ./a.out
Enter value of x = 10
Enter value of y = 2
sum is 0
-432912432 and 32651 values of x and y
Jacobs-MacBook-Pro:Parallel-and-Distributed-Computing jacobjohn$
```

Problem 2

Write a simple OpenMP program that uses some OpenMP API functions to extract information about the environment. It should be helpful to understand the languagecompiler features of OpenMP runtime library.

To examine the above scenario, the functions such as `omp_get_num_procs()`, `omp_set_num_threads()`, `omp_get_num_threads()`, `omp_in_parallel()`, `omp_get_dynamic()` and `omp_get_nested()` are listed and the explanation is given below to explore the concept practically:

- `omp_set_num_threads()` takes an integer argument and requests that the Operating System provide that number of threads in subsequent parallel regions.
- `omp_get_num_threads()` (integer function) returns the actual number of threads in the current team of threads.
- `omp_get_thread_num()` (integer function) returns the ID of a thread, where the ID ranges from 0 to the number of threads minus 1. The thread with the ID of 0 is the master thread.
- `omp_get_num_procs()` returns the number of processors that are available when the function is called.
- `omp_get_dynamic()` returns a value that indicates if the number of threads available in subsequent parallel region can be adjusted by the run time.
- `omp_get_nested( )` returns a value that indicates if nested parallelism is enabled.

Explanation

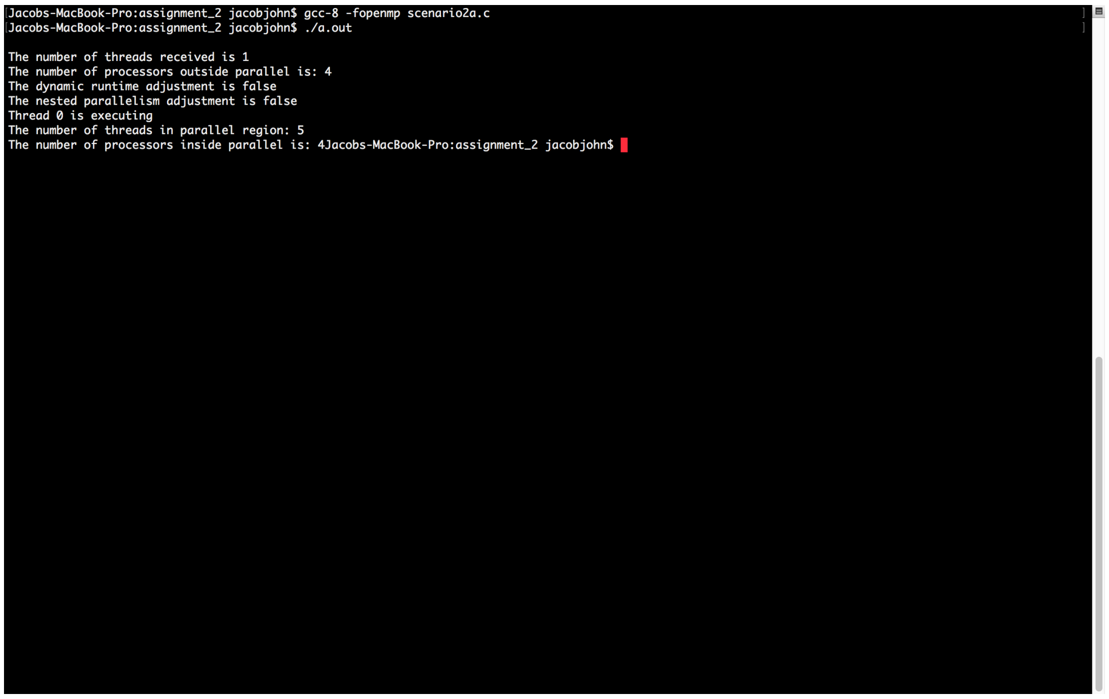
Using the Library classes available in openmp we can find out the environment conditions of the compiler when in use. Hence on running the code we see the environment conditions of the compiler in use.

The reason for the inefficiency for parallel search is given above. Each element will be checked regardless of a match. This is due to the fact that no thread can directly return after finding the element. Hence the longer execution time.

Listing 3: Scenario 2 using OpenMP

```c
#include <stdio.h>
#include <omp.h>
int main(){
    omp_set_num_threads(5);
    printf("\nThe number of threads received is %d",
    omp_get_num_threads());
    printf("\nThe number of processors outside parallel is: %d",
    omp_get_num_procs());
#pragma omp parallel
{
    int tid = omp_get_thread_num();
    if(tid == 0)
    {
        printf("\nThread %d is executing",
        tid);
        printf("\nThe number of threads in parallel region: %d",
        omp_get_num_threads());
        printf("\nThe number of processors inside parallel is: %d",
                omp_get_num_procs());
        }
        if(tid == 1)
            printf("\nThe dynamic runtime adjustment is %s",
                omp_get_dynamic() ? "true" : "false");
        if(tid == 2)
            printf("\nThe nested parallelism adjustment is %s",
                omp_get_nested() ? "true" : "false");
}
return 0;
}
```

Output:



```
Jacobs-MacBook-Pro:assignment_2 jacobjohn$ gcc-8 -fopenmp scenario2a.c
Jacobs-MacBook-Pro:assignment_2 jacobjohn$ ./a.out

The number of threads received is 1
The number of processors outside parallel is: 4
The dynamic runtime adjustment is false
The nested parallelism adjustment is false
Thread 0 is executing
The number of threads in parallel region: 5
The number of processors inside parallel is: 4Jacobs-MacBook-Pro:assignment_2 jacobjohn$
```

Problem 3

By using the *private()* and *shared()* directives, you can specify variables within the parallel region as being shared, i.e. visible and accessible by all threads simultaneously, or private, i.e. private to each thread, meaning each thread will have its own local copy. Write a simple OpenMP program that uses loop iteration counters in OpenMP loop. It constructs the for loop to be privately accessed variable (in case of executing the i variable). It is helpful to understand how threads make use of the cores to enable course-grain parallelism.

Explanation

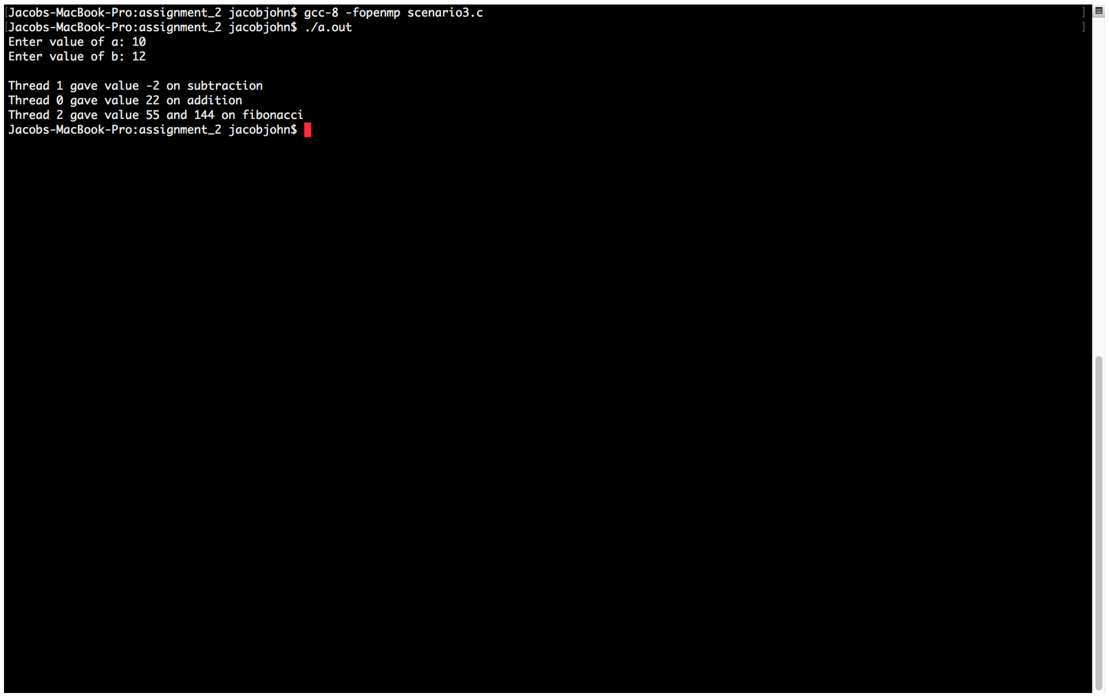
Shared variables are all declared variables outside the block declared as parallel, private variables are those where we need every thread to have a unique instance of the declared variable. Variables declared within the parallel scope are by default private and hence can work independently.

Therefore, we see the 3 threads run concurrently with the same shared values of a and b but all have different ID values. This is because the values a and b are declared as shared while the ID value is declared as private.

Listing 4: Scenario 3 using OpenMP in C

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

int add(int x, int y)
{
    return (x + y);
}

int sub(int x, int y)
{
    return (x - y);
}

int fib(int x)
{
    if (x <= 1)
        return x;
    return fib(x - 1) + fib(x - 2);
}

int main()
{
    int a,b;
    printf("Enter value of a: ");
    scanf("%d", &a);
    printf("Enter value of b: ");
    scanf("%d", &b);
    printf("\n");
    omp_set_num_threads(3);
#pragma omp parallel shared(a, b)
{
    int tid = omp_get_thread_num();
    if (tid == 0)
        printf("Thread %d gave value %d on addition\n", tid, add(a, b));
    if (tid == 1)
        printf("Thread %d gave value %d on subtraction\n", tid, sub(a, b));
    if (tid == 2)
        printf("Thread %d gave value %d and %d on fibonacci\n", tid, fib(a), fib(b));
}
return 0;
}
```

Output:



```
Jacobs-MacBook-Pro:assignment_2 jacobjohn$ gcc-8 -fopenmp scenario3.c
Jacobs-MacBook-Pro:assignment_2 jacobjohn$ ./a.out
Enter value of a: 10
Enter value of b: 12

Thread 1 gave value -2 on subtraction
Thread 0 gave value 22 on addition
Thread 2 gave value 55 and 144 on fibonacci
Jacobs-MacBook-Pro:assignment_2 jacobjohn$
```