University of Wolverhampton

School of Mathematics and Computer Science

6CS005 High Performance Computing Week 6 Workshop

Tasks - OpenMP Multithreading

Q1.

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ cat 1.c
#include <stdio.h>
#include <omp.h>

int main() {
    #pragma omp parallel
    printf("Hello from thread %d\n", omp_get_thread_num());
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ gcc 1.c -o 1
```

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ ./1
Hello from thread 15
Hello from thread 12
Hello from thread 5
Hello from thread 19
Hello from thread 11
Hello from thread 0
Hello from thread 18
Hello from thread 1
Hello from thread 14
Hello from thread 4
Hello from thread 2
Hello from thread 3
Hello from thread 6
Hello from thread 10
Hello from thread 13
Hello from thread 16
Hello from thread 9
Hello from thread 7
Hello from thread 17
Hello from thread 8
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$
```

This program creates an OpenMP parallel region where each thread prints its own thread number. Since the program does not specify how many threads to use, OpenMP automatically

launches one thread per CPU core. The output appears in a mixed order because all threads are running at the same time and printing independently.

Q2.

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ cat 2.c
#include <stdio.h>
#include <omp.h>

int main() {
    omp_set_num_threads(4);
    #pragma omp parallel
    printf("%d\n", omp_get_thread_num());
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ gcc -fopenmp 2.c -o 2
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ ./2
0
2
1
3
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$
```

This program sets the number of OpenMP threads to four before running the parallel section. Each thread prints its ID, so the output only shows thread numbers zero through three. Because the threads execute concurrently, the print statements appear in a jumbled order.

Q3.

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ cat 3.c
#include <stdio.h>
#include <omp.h>

int main() {
    #pragma omp parallel for
    for(int i=0;i<10;i++)
        printf("%d\n", i);
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ gcc -fopenmp 3.c -o 3
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ ./3
8
6
5
7
2
4
0
1
3
9
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$
```

This program runs a for loop in parallel, where different threads handle different loop iterations. Each iteration prints the current loop value, but the numbers do not appear in sequence because multiple threads are working simultaneously. The out-of-order output shows how parallel execution affects loop printing.

Q4.

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ cat 4.c
#include <stdio.h>
#include <omp.h>

int main() {
    int s=0;
    #pragma omp parallel for reduction(+:s)
    for(int i=1;i<=100;i++) s+=i;
    printf("%d\n", s);
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ gcc -fopenmp 4.c -o 4
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ ./4
5050
```

This program calculates the sum of numbers from one to one hundred using a parallel loop with a reduction. Each thread adds part of the total, and OpenMP safely combines these partial sums at the end. The program prints the correct final result because the reduction prevents threads from interfering with each other.

Q5.

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ cat 5.c
#include <stdio.h>
#include <omp.h>

int main() {
    int x=0;
    #pragma omp parallel
    {
        #pragma omp critical
        x++;
    }
    printf("%d\n", x);
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ gcc -focpenmp 5.c -o 5
gcc: error: unrecognized command-line option '-focpenmp'; did you mean '-fopenmp'?
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ gcc -fopenmp 5.c -o 5
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ ./5
20
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$
```

This program runs a parallel region where each thread increments a shared variable inside an OpenMP critical section. The critical section forces the threads to enter one at a time, so the updates happen safely. Because only one thread can access the section at once, the final printed

value increases slowly and ends up being twenty. The output reflects controlled access to the shared variable.

Q6.

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ cat 6.c
#include <stdio.h>
#include <omp.h>

int main() {
    int x=0;
    #pragma omp parallel
    {
        #pragma omp atomic
        x++;
    }
    printf("%d\n", x);
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ gcc -fopenmp 6.c -o 6
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ ./6
20
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$
```

This program performs the same shared update as Q5, but uses an OpenMP atomic operation instead of a critical section. The atomic directive ensures that each increment happens safely and without interference between threads. The final result is again twenty because all increments are applied correctly. Atomic is faster than critical but still prevents race conditions.

Q7.

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ cat 7.c
#include <stdio.h>
#include <omp.h>

int main() {
    int a=5;
    #pragma omp parallel private(a)
    printf("%d\n", a);
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ gcc -fopenmp 7.c -o 7
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ ./7
29033
29033
29033
29033
29033
29033
29033
29033
29033
29033
29033
29033
29033
29033
29033
29033
29033
29033
29033
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$
```

This program creates a parallel region where each thread prints the value of a private variable. Since the variable is private, each thread gets its own separate copy with the same starting value. The output shows the same number repeated many times because each thread prints its own version of the variable. No interference occurs because the variable is not shared.

Q8.

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ cat 8.c
#include <stdio.h>
#include <omp.h>

int main() {
    int a=5;
    #pragma omp parallel firstprivate(a)
    printf("%d\n", a+1);
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ gcc -fopenmp 8.c -o 8
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ ./8
6
6
6
6
6
6
6
6
6
6
6
6
6
6
6
6
6
6
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$
```

This program uses the firstprivate clause so each thread begins with a copy of the initial value of a variable. All threads print the same starting value because firstprivate duplicates the data for each one. The output shows identical numbers printed repeatedly. This demonstrates how firstprivate keeps original values separate for each thread.

Q9.

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ cat 9.c
#include <stdio.h>
#include <omp.h>

int main() {
    int x=0;
    #pragma omp parallel shared(x)
    printf("%d\n", x);
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ gcc -fopenmp 9.c -o 9
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ ./9
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$
```

This program uses the shared clause so all threads refer to the exact same variable during the parallel region. Since no thread modifies it, they all print the same initial value. The repeated zeros in the output show that each thread is reading from the same shared location. This contrasts with private and first private behavior.

Q10.

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ cat 10.c
#include <stdio.h>
#include <omp.h>

int main() {
    #pragma omp parallel
    {
        #pragma omp barrier
        printf("After barrier %d\n", omp_get_thread_num());
    }
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ gcc -fopenmp 10.c -o 10
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ ./10
After barrier 16
After barrier 12
After barrier 17
After barrier 19
After barrier 15
After barrier 8
After barrier 7
After barrier 13
After barrier 2
After barrier 6
After barrier 3
After barrier 5
After barrier 0
After barrier 18
After barrier 14
After barrier 4
After barrier 1
After barrier 10
After barrier 11
After barrier 9
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$
```

This program places a barrier inside the parallel region, forcing all threads to stop and wait at that point. After passing the barrier, each thread prints its ID, which is why the output shows “After barrier” followed by different thread numbers. The mixed order comes from threads reaching and leaving the barrier at slightly different times. The barrier ensures synchronization among the threads.

Q11.

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ cat 11.c
#include <stdio.h>
#include <omp.h>

int main() {
    #pragma omp parallel
    {
        #pragma omp master
        printf("Master: %d\n", omp_get_thread_num());
    }
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ gcc -fopenmp 11.c -o 11
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ ./11
Master: 0
```

This program uses the OpenMP master directive so only the master thread executes the print statement. The other threads skip that part entirely. The output shows only one line stating the master thread's ID. This behavior demonstrates how the master directive restricts execution to a single specific thread.

Q12.

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ cat 12.c
#include <stdio.h>
#include <omp.h>

int main() {
    #pragma omp parallel
    {
        #pragma omp single
        printf("Only once\n");
    }
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ gcc -fopenmp 12.c -o 12
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ ./12
Only once
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$
```

This program uses the OpenMP single directive, which allows only one of the threads to run the enclosed code block. Unlike master, any thread can become the one that executes it. The output prints a single message even though multiple threads are running. This shows how single selects only one thread to perform a task.

Q13.

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ cat 13.c
#include <stdio.h>
#include <omp.h>

int main() {
    #pragma omp parallel sections
    {
        #pragma omp section
        printf("A\n");
        #pragma omp section
        printf("B\n");
    }
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ gcc -fopenmp 13.c -o 13
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ ./13
B
A
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$
```

This program divides the work into separate sections so different threads execute different blocks of code. One section prints “A” while another prints “B”, and the order depends on which thread finishes first. The output shows both letters but not necessarily in sequence. This demonstrates OpenMP’s ability to run independent tasks in parallel.

Q14.

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ cat 14.c
#include <stdio.h>
#include <omp.h>

int main() {
    #pragma omp parallel for schedule(dynamic,2)
    for(int i=0;i<10;i++)
        printf("%d\n", i);
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ gcc -fopenmp 14.c -o 14
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ ./14
2
8
0
1
9
6
7
4
5
3
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$
```

This program parallelizes a loop using dynamic scheduling, which assigns chunks of work to threads as they become free. The output shows loop values printed in a mixed order because threads handle the iterations at different speeds. Dynamic scheduling helps balance the workload when iterations take varying amounts of time. The output reflects this uneven distribution.

Q15.

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ cat 15.c
#include <stdio.h>
#include <omp.h>

int main() {
    omp_set_nested(1);
    #pragma omp parallel num_threads(2)
    {
        #pragma omp parallel num_threads(2)
        printf("%d\n", omp_get_thread_num());
    }
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ gcc -fopenmp 15.c -o 15
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ ./15
0
1
1
0
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$
```

This program sets the number of OpenMP threads using omp set num threads and then prints each thread number inside a parallel region. The result shows several thread IDs depending on the system and environment. The repeated lines indicate multiple threads executing the print statement. This demonstrates programmatic control over thread count.

Q16.

```
98561
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ cat 16.c
#include <stdio.h>
#include <omp.h>

int isprime(int n) {
    if(n<2) return 0;
    for(int i=2;i*i<=n;i++)
        if(n%i==0) return 0;
    return 1;
}

int main() {
    #pragma omp parallel for schedule(dynamic)
    for(int i=1;i<=100000;i++)
        if(isprime(i))
            printf("%d\n", i);
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$
```

This program uses a function to test whether numbers are prime and processes a large range in parallel. Because the loop is scheduled dynamically, threads print numbers whenever they finish checking them. The output appears in an unpredictable order due to varying execution times of each prime test. This illustrates how dynamic scheduling affects real workloads.

Q17.

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ cat 17.c
#include <stdio.h>
#include <omp.h>

int isprime(int n) {
    if(n<2) return 0;
    for(int i=2;i*i<=n;i++)
        if(n%i==0) return 0;
    return 1;
}

void p1() {
    for(int i=1;i<=50000;i++)
        if(isprime(i))
            printf("%d\n", i);
}

void p2() {
    for(int i=50001;i<=100000;i++)
        if(isprime(i))
            printf("%d\n", i);
}

int main() {
    #pragma omp parallel sections
    {
        #pragma omp section
        p1();
        #pragma omp section
        p2();
    }
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$
```

This program uses parallel sections to run two separate functions at the same time. Each function processes a range of numbers and prints whenever a prime is found. The mixed output comes from both sections running concurrently. This shows how OpenMP can execute independent computations simultaneously.

Q18.

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ cat 18.c
#include <stdio.h>
#include <omp.h>

int isprime(int n) {
    if(n<2) return 0;
    for(int i=2;i*i<=n;i++)
        if(n%i==0) return 0;
    return 1;
}

int main() {
    int n;
    int count=0;
    scanf("%d",&n);

    #pragma omp parallel for reduction(+:count)
    for(int i=1;i<=n;i++)
        if(isprime(i)) count++;

    if(count>500) {
        omp_set_num_threads(2);
        #pragma omp parallel for schedule(dynamic)
        for(int i=1;i<=n;i++)
            if(isprime(i))
                printf("%d\n", i);
    } else {
        #pragma omp parallel for
        for(int i=1;i<=n;i++)
            if(isprime(i))
                printf("%d\n", i);
    }
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$
```

```
^[[A^C
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$ ./18
20
13
17
5
7
2
11
3
19
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6/Week6/openmp_class$
```

This program performs a reduction to count how many numbers in a range are prime and then splits the work differently depending on that count. If the number of primes exceeds five hundred, the program uses two threads and dynamic scheduling; otherwise it uses a single thread. The printed results show how many primes were found and which thread handled each range. This demonstrates conditional parallel behavior based on computed values.