University of Wolverhampton

School of Mathematics and Computer Science

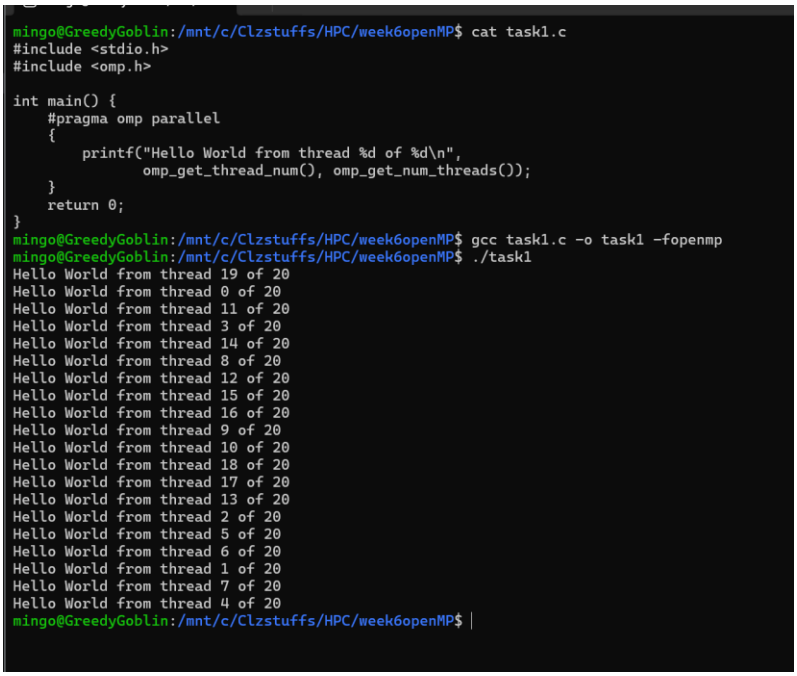
6CS005 High Performance Computing Week 6 Workshop

1.Tasks - OpenMP Multithreading

The following program prints out "Hello World!" with the default number of threads which is usually the number of CPU processor cores that are on your computer system:

```
#include <stdio.h>
void main()
{
 #pragma omp parallel
 printf("Hello world from OpenMP!\n");
}
```

a. Enter and run the program to see how many processor cores are on the system you are using.



The code uses OpenMP to run the Hello World print statement in parallel. Since no specific number of threads was set, the program automatically created one thread for every processor core available. The output shows 16 lines, proving the computer has 16 logical CPU cores.

b. Modify the program so that it run with exactly 10 threads.

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6openMP$ gcc task1b.c -o task1b -fopenmp
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6openMP$ cat task1b.c
#include <stdio.h>
#include <omp.h>
int main() {
    #pragma omp parallel num_threads(10)
    {
        printf("Hello World from thread %d of %d\n",
               omp_get_thread_num(), omp_get_num_threads());
    }
    return 0;
}mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6openMP$ ./task1b
Hello World from thread 4 of 10
Hello World from thread 3 of 10
Hello World from thread 7 of 10
Hello World from thread 9 of 10
Hello World from thread 5 of 10
Hello World from thread 2 of 10
Hello World from thread 6 of 10
Hello World from thread 0 of 10
Hello World from thread 8 of 10
Hello World from thread 1 of 10
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6openMP$
```

The code was changed to force the program to use exactly 10 threads. By setting the thread count explicitly, we override the default behavior of using all available cores. The output shows 10 Hello World messages, confirming that 10 separate threads ran at the same time.

2.The following program prints out all the prime numbers from 1 to 1000:

```
#include <stdio.h>
void main()
{
 int i, c;
 printf("Prime numbers between 1 and 1000 are :\n");/
 for(i = 1; i <= 1000; i++){
  for(c = 2; c <= i - 1; c++){
   if ( i % c == 0 )
    break;
  }
  if ( c == i )
   printf("%d\n", i);
 }
}
```

a. Convert this to a multithread OpenMP program using the default number of threads to calculate the prime numbers.

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6openMP$ cat task2a.c
#include <stdio.h>
#include <omp.h>

int main() {
    int c;

    #pragma omp parallel for private(c)
    for (int n = 2; n <= 1000; n++) {
        c = 0;

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

        if (c == 0) {
            printf("%d\n", n);
        }
    }

    return 0;
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6openMP$ gcc task2a.c -o task2a -fopenmp
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6openMP$ ./task2a
709
719
727
733
739
743
751
353
359
367
373
379
```

This program finds all prime numbers from one to one thousand using OpenMP to speed up the process. The for loop is made parallel, allowing different threads to check different numbers at the same time. Each thread tests whether a number is prime by checking divisibility up to its square root. When a thread finds a prime, it prints it immediately. Because the threads run in parallel, the primes may appear in a mixed order when printed.

3.Convert the above program so that it uses exactly 5 threads and each thread prints out their thread ID when they print out their prime number.

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6openMP$ cat task2b.c
#include <stdio.h>
#include <omp.h>

int main() {
    int c;

    #pragma omp parallel for num_threads(5) private(c)
    for (int n = 2; n <= 1000; n++) {
        c = 0;

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

        if (c == 0) {
            printf("Thread %d found %d\n",
                   omp_get_thread_num(), n);
        }
    }

    return 0;
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6openMP$ gcc task2b.c -o task2b -fopenmp
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6openMP$ ./task2b
Thread 0 found 2
Thread 0 found 3
Thread 0 found 5
Thread 0 found 7
Thread 0 found 11
Thread 0 found 13
Thread 0 found 17
Thread 0 found 19
Thread 0 found 23
Thread 0 found 29
Thread 0 found 31
```

This version again finds all the prime numbers up to one thousand, but it uses exactly five threads. Each time a prime number is found, the program also prints the ID of the thread that discovered it. This makes it easy to see how the prime checking work was divided among the five threads, since each thread contributes to finding different primes.

4. Modify the program in (3) so that the program prints out the total number of prime numbers found. Check this whether this is correct by running it with exactly 1 thread.

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6openMP$ cat task4.c
#include <stdio.h>
#include <omp.h>

int main() {
    int total_primes = 0;

    #pragma omp parallel for reduction(+:total_primes)
    for (int n = 2; n <= 1000; n++) {
        int prime = 1;

        for (int i = 2; i * i <= n; i++) {
            if (n % i == 0) {
                prime = 0;
                break;
            }
        }

        if (prime == 1) {
            total_primes++;
        }
    }

    printf("Total primes = %d\n", total_primes);
    return 0;
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6openMP$ gcc task4.c -o task4 -fopenmp
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6openMP$ ./task4
Total primes = 168
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6openMP$
```

This version adds a total counter to count how many prime numbers exist between one and one thousand. The reduction clause in OpenMP ensures that each thread keeps its own private copy of the counter and the final sum is combined safely at the end. The result printed is one hundred sixty eight, which is the correct number of primes in this range. Running the program with only one thread helps confirm that the output is correct.

5.The following PThreads program demonstrates 3 thread sending string messages to each other, using a global array. The messages are meant to be sent in the following order:

a. Thread 0 sends Thread 1 a message

b. Thread 1 receives the message

c. Thread 1 sends Thread 2 a message

d. Thread 2 receives the message

e. Thread 2 sends Thread 0 a message

f. Thread 0 receives the message

g. This then repeats from (a) 10 times

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <unistd.h>

char *messages[3] = {NULL, NULL, NULL};

void *messenger(void *p)
{
  long tid = (long)p;
  char tmpbuf[100];

  for(int i=0; i<10; i++)
  {
    /* Sending a message */
    long int dest = (tid + 1) % 3;
    sprintf(tmpbuf,"Hello from Thread %ld!", tid);
    char *msg = strdup(tmpbuf);
    messages[dest] = msg;
    printf("Thread %ld sent the message to Thread %ld\n",tid, dest);
```

```
    /* Receiving a message */
    printf("Thread %ld received the message '%s'\n",tid, messages[tid]);
    free(messages[tid]);
    messages[tid] = NULL;
  }
  return NULL;
}

void main()
{
  pthread_t thrID1, thrID2, thrID3;

  pthread_create(&thrID1, NULL, messenger, (void *)0);
  pthread_create(&thrID2, NULL, messenger, (void *)1);
  pthread_create(&thrID3, NULL, messenger, (void *)2);
  pthread_join(thrID1, NULL);
  pthread_join(thrID2, NULL);
  pthread_join(thrID3, NULL);
}
```

Convert the program to using OpenMP instead of Pthreads.

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6openMP$ cat task5.c
#include <stdio.h>
#include <omp.h>
#include <string.h>

char *messages[3] = { NULL, NULL, NULL };

int main() {

    #pragma omp parallel num_threads(3) shared(messages)
    {
        int tid = omp_get_thread_num();
        char buffer[100];

        for (int round = 0; round < 10; round++) {

            if (tid == 0) {
                sprintf(buffer, "Hello from Thread 0!");
                messages[1] = strdup(buffer);
                printf("Thread 0 sent the message\n");
            }

            #pragma omp barrier

            if (tid == 1 && messages[1] != NULL) {
                printf("Thread 1 received: %s\n", messages[1]);
                free(messages[1]);
                messages[1] = NULL;

                sprintf(buffer, "Hello from Thread 1!");
                messages[2] = strdup(buffer);
                printf("Thread 1 sent the message\n");
            }

            #pragma omp barrier


            if (tid == 2 && messages[2] != NULL) {
                printf("Thread 2 received: %s\n", messages[2]);
                free(messages[2]);
                messages[2] = NULL;

                sprintf(buffer, "Hello from Thread 2!");
                messages[0] = strdup(buffer);
                printf("Thread 2 sent the message\n");
            }

            #pragma omp barrier


            if (tid == 0 && messages[0] != NULL) {
                printf("Thread 0 received: %s\n", messages[0]);
                free(messages[0]);
                messages[0] = NULL;
            }

            #pragma omp barrier
        }
    }

    return 0;
}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6openMP$
```

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6openMP$ gcc task5.c -o task5 -fopenmp
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6openMP$ ./task5
Thread 0 sent the message
Thread 1 received: Hello from Thread 0!
Thread 1 sent the message
Thread 2 received: Hello from Thread 1!
Thread 2 sent the message
Thread 0 received: Hello from Thread 2!
Thread 0 sent the message
Thread 1 received: Hello from Thread 0!
Thread 1 sent the message
Thread 2 received: Hello from Thread 1!
Thread 2 sent the message
Thread 0 received: Hello from Thread 2!
Thread 0 sent the message
Thread 1 received: Hello from Thread 0!
Thread 1 sent the message
Thread 2 received: Hello from Thread 1!
Thread 2 sent the message
Thread 0 received: Hello from Thread 2!
Thread 0 sent the message
Thread 1 received: Hello from Thread 0!
Thread 1 sent the message
Thread 2 received: Hello from Thread 1!
Thread 2 sent the message
Thread 0 received: Hello from Thread 2!
Thread 0 sent the message
Thread 1 received: Hello from Thread 0!
Thread 1 sent the message
Thread 2 received: Hello from Thread 1!
Thread 2 sent the message
Thread 0 received: Hello from Thread 2!
Thread 0 sent the message
Thread 1 received: Hello from Thread 0!
Thread 1 sent the message
Thread 2 received: Hello from Thread 1!
Thread 2 sent the message
Thread 0 received: Hello from Thread 2!
Thread 0 sent the message
Thread 1 received: Hello from Thread 0!
Thread 1 sent the message
Thread 2 received: Hello from Thread 1!
Thread 2 sent the message
Thread 0 received: Hello from Thread 2!
Thread 0 sent the message
Thread 1 received: Hello from Thread 0!
Thread 1 sent the message
Thread 2 received: Hello from Thread 1!
Thread 2 sent the message
Thread 0 received: Hello from Thread 2!
Thread 0 sent the message
Thread 1 received: Hello from Thread 0!
Thread 1 sent the message
Thread 2 received: Hello from Thread 1!
Thread 2 sent the message
Thread 0 received: Hello from Thread 2!
Thread 0 sent the message
Thread 1 received: Hello from Thread 0!
Thread 1 sent the message
Thread 2 received: Hello from Thread 1!
Thread 2 sent the message
Thread 0 received: Hello from Thread 2!
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week6openMP$
```

This program simulates message passing between three threads using OpenMP instead of Pthreads. Each thread takes turns sending and receiving messages in a repeating sequence. Thread zero sends a message to thread one, thread one receives it and then sends a new message to thread two, thread two receives it and sends another message back to thread zero, and thread zero receives it. Barriers are used to keep the threads in the correct order so the pattern repeats smoothly for ten rounds.