BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI (RAJASTHAN)

CS F422 – Parallel Computing

Lab#2

Note: Please use programs under *Code_lab2* directory supplied with this sheet. Do not copy from this sheet.

The lab has the following objectives:
Giving practice programs for OpenMP and OpenCilk.

Compiling and Running an OpenMP Program

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

int main(int argc, char* argv[]) {
    printf("Hello World\n");
    printf("No. of parallel process possible: %d\n", omp_get_num_procs());
#pragma omp parallel
    {
        // printf("I am a parallel region.\n");
              printf("Hi I'm parallel process no. : %d\n", omp_get_thread_num());
          }
          return 0;
      }
```

Q?

1. Run the program with `` `gcc openmp_create.c -fopenmp` ``. How is the program deciding the number of parallel processes?
2. Find out if it's possible to change the number of available parallel processes for the program.

Monitoring an OpenMP Program

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

long fib(int n) {
    return (n < 2 ? 1 : fib(n - 1) + fib(n - 2));
}

int main(int argc, char* argv[]) {
    int n = 42;
      #pragma omp parallel
          {
              int t = omp_get_thread_num();
              printf("%d: %ld\n", t, fib(n + t));
          }
          return 0;
      }
```

Q?

1. Run the program with `env OMP_NUM_THREADS=8 time ./a.out`.
2. Real time means the actual or "wall-clock" time taken. The user time is the time taken together for all logical cores to run their respective program. Compare the two numbers. Why is one bigger than the other? Is it expected to be this way?

Parallelization of Loops

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

int main(int argc, char* argv[]) {
    int max;
    sscanf(argv[1], "%d", &max);
#pragma omp parallel for
    for (int i = 1; i <= max; i++)
        printf("%d : (%d ,%d)\n ", omp_get_thread_num(), i);
          return 0;
      }
```

Q?

1. Run the above program with `` `./a.out 10` ``.
2. Try to create a nested loop by adding a j sequence from 1 to max. Run it a few times. Based on your output, try to understand how OpenMP is parallelizing this loop.
3. How can you convert the nested loop into parallelizing all printf statements?

OpenMP: nested parallelism

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

#define N 3
#define MAX_NUM 10

void mtxMul(int c[][N], int a[][N], int b[][N], int n) {
#pragma omp parallel for collapse(2)
          for (int i = 0; i < n; i++)
              for (int j = 0; j < n; j++) {
                  c[i][j] = 0;
                  for (int k = 0; k < n; k++)
                      c[i][j] += a[i][k] * b[k][j];
              }
          return;
      }

      void display(int result[][N], int n) {
          printf("\nOutput Matrix:\n");
          for (int i = 0; i < n; ++i) {
              for (int j = 0; j < n; ++j) {
                  printf("%d  ", result[i][j]);
                  if (j == n - 1)
                      printf("\n");
              }
          }
      }

      int main(int argc, char* argv[]) {
          int arr1[N][N], arr2[N][N], mult[N][N];
          for (int i = 0; i < N; ++i) {
              for (int j = 0; j < N; ++j) {
                  arr1[i][j] = rand() % MAX_NUM;
                  arr2[i][j] = rand() % MAX_NUM;
              }
37.         }
38.         mtxMul(mult, arr1, arr2, N);
39.         display(arr1, N);
40.         display(arr2, N);
41.         display(mult, N);
42.         return 0;
43.     }
```

Q?

1. Run the above program. Change the size N to a bigger value (~500).
2. Now remove the pragma line at Line 9. Observe the difference in time taken.

Combining the Results of Parallel Iterations

```
1. #include <omp.h>
2. #include <stdio.h>
3.
4. int main(int argc, char* argv[]) {
5.     int max, sum = 0;
6.     sscanf(argv[1], "%d", &max);
7. #pragma omp parallel for
8.     for (int i = 1; i <= max; i++)
9. #pragma omp atomic
10.            sum += i;
11.        printf("%d\n", sum);
12.        return 0;
13.    }
```

Q?

1. Run the above program `./a.out 1000000`.
2. Observe how faster this is compared to critical sections (below).

OpenMP: critical sections

```
1. #include <omp.h>
2. #include <stdio.h>
3.
4. int main(int argc, char* argv[]) {
5.     int max, sum = 0;
6.     sscanf(argv[1], "%d", &max);
7. #pragma omp parallel for
8.     for (int i = 1; i <= max; i++)
9. #pragma omp critical
```

```
              sum += i;
          printf("%d\n", sum);
          return 0;
      }
```

Q?

1. Run the above program `` `./a.out 1000000` ``.
2. Try removing the `#pragma omp critical`. Do you observe any unexpected output? Why is this?

Distributing Iterations Among Thread

```
1. #include <omp.h>
2. #include <stdio.h>
3. #include <unistd.h>
4.
5. int main(int argc, char* argv[]) {
6.     long int max, sum = 0;
7.     sscanf(argv[1], "%ld", &max);
8. #pragma omp parallel for reduction (+:sum) schedule(runtime)
9.     for (int i = 1; i <= max; i++) {
10.              printf("%2d @ %d\n", i, omp_get_thread_num());
11.              sleep(i < 4 ? i + 1 : 1);
12.              sum += i;
13.          }
14.          printf("%ld\n", sum);
15.          return 0;
16.      }
```

Q?

1. Run the above program `` `env OMP_SCHEDULE=static ./a.out 10` ``.
2. Change OMP_SCHEDULE to dynamic and see the difference in output.
3. Specify the chunk size for schedule in Line 8.

Parallel Tasks

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

int main(int argc, char* argv[]) {
    int max;
    sscanf(argv[1], "%d", &max);
    int tasks;
    sscanf(argv[2], "%d", &tasks);
```

```

  if (max % tasks != 0) return 1;
  int sum = 0;

#pragma omp parallel
  {
#pragma omp single
    for (int t = 0; t < tasks;++t) {
#pragma omp task
      {
        int local_sum = 0;
        int lo = (max / tasks) * (t + 0) + 1;
        int hi = (max / tasks) * (t + 1) + 0;
        for (int i = lo;i <= hi;++i) {
          local_sum += i;
        }
#pragma omp atomic
        sum += local_sum;
      }
    }
  }
  printf("%d\n", sum);
  return 0;
}
```

Q?

1. Run the above program `./a.out 10000 10`.
2. Observe how the program is divided into independent tasks, in contrast to parallelizing loops so far.

End of lab1