**BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI (RAJASTHAN)**

**CS G532 – High Performance Heterogeneous Computing**

**Lab#4**

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

The lab has the following objectives:

Giving practice programs for OpenMP.

**Compiling and Running an OpenMP Program**

1. #include <omp.h>
2. #include <stdio.h>
3. int main(int argc, char\* argv[]) {
4. printf("Hello World\n");
5. printf("No. of parallel process possible: %d\n", omp\_get\_num\_procs());
6. #pragma omp parallel
7. {
8. // printf("I am a parallel region.\n");
9. printf("Hi I'm parallel process no. : %d\n", omp\_get\_thread\_num());
10. }
11. return 0;
12. }

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**

1. #include <omp.h>
2. #include <stdio.h>
3. long fib(int n) {
4. return (n < 2 ? 1 : fib(n - 1) + fib(n - 2));
5. }
6. int main(int argc, char\* argv[]) {
7. int n = 42;
8. #pragma omp parallel
9. {
10. int t = omp\_get\_thread\_num();
11. printf("%d: %ld\n", t, fib(n + t));
12. }
13. return 0;
14. }

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?

**Scope of Variables and reduction clause**

Refer to the program given in omp\_trap1.c. This program computes integral of a function given its intervals. Trap function is parallelized using the following directive.

# pragma omp parallel num\_threads(thread\_count)

Trap(a, b, n, &global\_result);

Q?

1. Check out whether this will give correct results. Each thread is adding to global\_ result variable. Modify the code to make sure that global\_result variable is protected.

You can use:

# pragma omp critical

\*global\_result\_p += my\_result;

1. If we change the parallel directive in the following manner, will it improve performance? Explain. You can check the time usage using $time ./a.out

global\_result =0.0;

# pragma omp parallel num\_threads(thread\_count)

{

# pragma omp critical

global\_result += Local\_trap(a,b,n);

}

1. If we change the parallel directive in the following manner, will it improve performance? Explain. You can check the time usage using $time ./a.out

global\_result = 0.0;

# pragma omp parallel num\_threads(thread\_count)

{

double my\_result = 0.0;

my\_result += Local\_trap(a, b, n);

# pragma omp critical

global\_result += my\_result;

}

1. OpenMP provides a cleaner alternative in the form of reduction clause. If we change the parallel directive in the following manner, will it improve performance? Explain. You can check the time usage using $time ./a.out

# pragma omp parallel num\_threads(thread\_count) \

reduction(+: global\_result)

global\_result += Local\_trap(a, b, n);

1. If we use parallel for to parallelize loops instead of function, will it improve performance? Can you tell work done in iterations is uniform or not? Which is the best scheduling class and chunk partitioning in this case?

double Trap(double a, double b, int n, int thread\_count) {

double h, approx;

int i;

h = (b-a)/n;

approx = (f(a) + f(b))/2.0;

# pragma omp parallel for num\_threads(thread\_count) \

reduction(+: approx)

for (i = 1; i <= n-1; i++)

approx += f(a + i\*h);

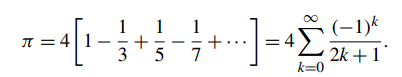
approx = h\*approx;

return approx;

} /\* Trap \*/

**Loop-carried Dependencies**

Consider the given code in file omp\_pi.c. This program estimates the following series.



1. for (i = 0; i < n; i++) {
2. sum += factor/(2\*i+1);
3. factor=-factor;
4. }

Q?

1. Does this loop carry a loop-carried dependence? How can we make it free from loop carried dependency? [Hint: factor = -factor can be eliminated]
2. If we compile and run the code given in omp\_pi.c, it will not give correct results. What could be wrong? [Hint: Check the scope of factor variable.]

**Odd-even Transposition Sort**

Consider the following code for odd-even transposition sort.

1. for (phase = 0; phase < n; phase++) {
2. if (phase % 2 == 0)
3. for (i = 1; i < n; i += 2) {
4. if (a[i-1] > a[i]) {
5. tmp = a[i-1];
6. a[i-1] = a[i];
7. a[i] = tmp;
8. }
9. }
10. else
11. for (i = 1; i < n-1; i += 2) {
12. if (a[i] > a[i+1]) {
13. tmp = a[i+1];
14. a[i+1] = a[i];
15. a[i] = tmp;
16. }
17. }

Q?

1. Do these two loops carry a loop-carried dependence? Which one can be parallelized? What is the problem in parallelizing outer loop?
2. Write *omp parallel for* statements in omp\_odd\_even.c file. What should be the scope of i? What should be the scope of tmp? Run the program for 1..6 threads and note the time taken.
3. If we write omp parallel once outside outer for loop and only write “#pragma omp for” for inner loops, will it reduce time. Justify.

**Scheduling Loops**

OpenMP provides several scheduling classes: static, dynamic, guided, runtime, auto.

Consider Sum() and f() in the file omp\_sin\_sum.c:

Q?

1. Run the program with static schedule with both block and cyclic partitioning and dynamic schedule and guided schedule. Record the time taken. Explain the difference.

Run the following program

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

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.

**Combining the Results of Parallel Iterations**

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

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

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?

**Lock Free Stack Implementation**

Lock free stack is implemented using CAS instructions. Code is given in lf\_stack.c and lf\_stack\_main.c files. Two threads push and pop from a stack a large number of times.

Q?

1. Compile lf\_stack\_main.c file and run. Observe the output produced. Is it always popping the last pushed value? How can you explain this inconsistent operations although using atomic CAS instruction?[Hint: ABA problem]
2. Can you rectify this problem by delaying the memory freeing? [Hint: Line 47 free() can be commented]

**End of lab4**