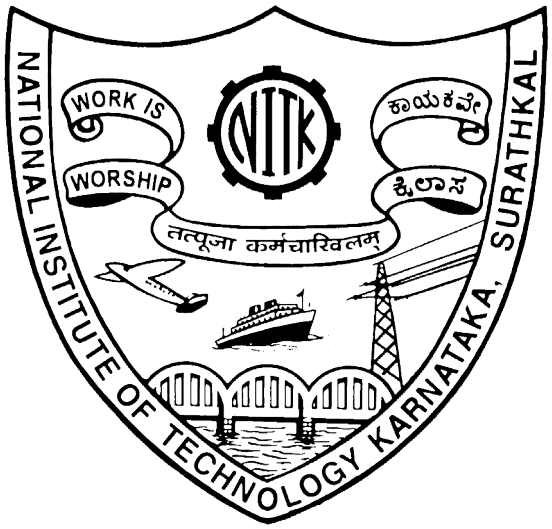
National Institute of Technology Karnataka
Computer Science and Engineering Department

Parallel Programming (CO471) Assignment 1
Open Multiprocessing (OpenMP)



Group Members

Vignesh K (14CO252)
V.B. Vineeth Reddy (14CO151)

Q1. Hello World

In this program, the master thread creates a parallel section with a default number of threads using the **#pragma omp parallel** directive and each thread prints the “Hello World” message.

Q2. Hello World with ID

In this program, each thread created in the parallel section retrieves its ID by calling the **omp_get_thread_num()** function and passes this as a parameter to a function. The function call prints the “Hello World” message with the ID of the thread which called the function.

Q3. DAXPY Loop

In this program, if we ignore minor fluctuations, the speedup intitially increases as we increase the number of threads. As the number of threads goes beyond 8, the speedup stagnates and finally takes a minor dip as we reach 16 threads. This can be attributed to the overhead of communication between threads and the existence of only 4 virtual cores. As we go beyond 4, the overhead starts dominating over the performance advantage attained by parallelization. Also the limit of 4 cores allows only 4 threads to actually run parallely. Another reasson for the stagnation is the shared L3 cache acting as a bottleneck for the communicating threads thus preventing the further increase of the speedup. The work-sharing construct **#pragma omp for** is used to distribute the computation of the DAXPY over the number of allocated threads.

Q4. Matrix Multiply

In this program, we have viewed the computation of each element of the ans[][] matrix as an independent task. So we have N x N independent tasks to parallelize. To parallelize these tasks we have converted the two nested loops that we usually use to iterate over matrices, into a single loop using the row-majpr formula. We then apply the work-sharing construct over this single loop to **parallelize the computation of each element which is esentially a dot-product** of a row with a column. Again as in the previous problem, the speedup increases till 4 threads, stagnates and then dips. This is due to the same reasons mentioned above. Let's take an example of N = 1000. We then have 1000000 elements whose computation we want to parallelize. But there are only 4 virtual cores. Clearly the parallelization we can achieve is limited.

Q5. Calculation of π

In this program, we use the **SPMD (Shared Program Multiple Data)** technique to compute the area under the curve. This is a **Round-Robin technique** where we explicitly assign subsets of datapoints to each thread which then computes the partial sum of the curve over these assigned datapoints. The partial sum variable is declared inside the parallel section which means that it is private to each thread. So we need not worry about synchronizing the updation of the partial sum variable. However, after each thread has completed computing its partial sum, this partial sum needs to be added to the PI variable which is shared among the threads because of being declared outside the parallel secion. We hence use the **#pragma omp atomic** construct to ensure that the updation of the PI variable is done as an atomic instruction without interference from other threads. This ensures that the updates to PI are consistent.

Q6. Calculation of π – Worksharing and Reduction

In this program, instead of explicitly assigning the share of work to each thread, we use the **#pragma omp for reduction (+ : sum)** construct. This automatically distributes the work among the available threads equally. The reduction construct applied on the sum variable over the '+' operator dictates that sum will be a private variable for each thread while inside the loop and the thread-specific values of sum will be put together using the '+' operator into the variable sum after the execution of the loop statement. So we now automatically get the value of PI into the sum variable once each thread has computed its respective partial sum.

Q7. Calculation of π - Monte Carlo Simulation

In this program, we randomly throw points on a square with a circle inscribed in it. By computing the ratio of the number of points which fall insde the circle to the total number of points thrown and multiplying it by 4, we get the value of PI. To parallelize the throwing of points, we nee each thread to generate a random number for the x and y coordinates. We use the LCG pseudo random number generator to do this. This generator computes the next random number based on the previous random number. We hence declare last_rand as a global variable to preserve the state. But just declaring globally is not enough because two or more threads may end up using the same last_rand value which renders the whole process as non-random. We thus make last_rand as **threadprivate.** Another problem is that if we randomly assign seed values to the threads, they may end up generating overlapping sequences of random number which again makes the process non-random. The solution we have used here is the famous **leap-frog method.** In this method, N threads are given the first N terms of the pseudo-random sequence as their respective seeds. Having done this, each thread jumps N steps from its last generated random

number to generate the next random number instead of 1 step as before. This ensures that the sequences generated by the different threads do not overlap. This is again similar to the Round-Robin technique.

Q8. Producer-Consumer Program

In this program, the objective is to synchronize the activities of a producer-consumer pair. In short, the consumer thread should execute its section only after the producer thread has executed its section. For this we use a combination of a flag variable and the **flush** functionality. The flag variable is initially set to 0 which indicates that the producer is yet to populate the array. To make the consumer thread wait till the population is complete, we have a while loop which keeps running as long as flag is 0. Once the producer has finished populating the array, we set flag to 1 to indicate this. The flush routine is then invoked by the producer thread so that the updated value of flag is written to the main memory. We also flush the populated array to the main memory for its values to be visible to other threads. Meanwhile, we also call the flush routine in the body of the waiting while loop of the consumer so that it reads the updated value of the flag variable and doesn't keep using its cache value of 0. So once the consumer sees that flag is 1, it exits the while loop. It then flushes again to read the updated array from the memory. Finally it sums the array.