A5-CO316 - Computer Architecture Lab report

Rana Hardik - 16CO138

Harshal Shinde - 16CO223

Q1. Hello World Program

We spawned the default number of threads were spawned to display the hello world message. The default number of threads came out to be 5.

Q2. Hello World Version-2

The spawned threads had to execute a function which had thread id as the argument. The function displayed the hello world message and thread id.

Q3. DAXPY

In the DAXPY loop the two Input arrays X and Y are updated as follows : X = A*X Using omp parallel for work distribution construct having x,y as shared data and i (loop index) as private variable the code was parallelised. The code was run taking number of threads from 2 and 10 and was also compared to the serialized implementation. In some cases serial code performed better than parallel code due to the work distribution overhead. But it's evident that as the number of threads increased parallel execution was better as compared to serial one. After a certain point, the speedup decreases because of Context Switching. When the number of threads increase, a lot of context switching takes place when compared to less number of threads.

7 threads gave the maximum speedup in most cases, after which there was seen a down-fall on machine with 4 cores and 4 GB RAM, i3 processor.

Q4. Matrix Multiply

The OpenMP work distribution construct omp parallel was used to do parallel computation. It was compared with the serial one. Number of threads spawned and the data elements they operate on are handled by OpenMP implicitly.

Q5. Calculation of Pi

We used 2 methods to calculate the value of pi. One to avoid false sharing we used padding to

compute the sum array and then store the value in the global pi variable. In the second method we used critical synchronization construct of OpenMP so that inside the parallel section of code we can update the value of pi and no race condition occurs. Since pi is a shared variable each thread tries to update but it can lead to data inconsistency by making it inside the critical block we ensure mutual exclusion.

Q6. Calculation of Pi using work-distribution

In this code we calculated the value of pi using omp for reduction (op:var) construct. Here the operation was + and variable was sum as all the threads were adding to the sum variable which was a shared variable. Later after the parallelised section of code pi value was updated.

Q7. Calculation of Pi using Monte-Carlo Simulation

In this code the value of pie is calculated using the monte-carlo simulation, the method followed is as follows: generate random numbers of precision double in range (0,1) and then count the number of such points generated which lie inside the circle. The number of points generated are 2^30. The ratio of count and number of points multiplied by 4 gives the value of pie. The function to create the random number reference was taken from here This random function generator uses the logic used in the rand() function in C/C++.

For the thread safe random function generator, the Generator is a linear congruential generator with constants selected to yield decent results for sequences with fewer than 2^28 numbers. The pseudo random sequence is seeded with a range.It uses the leapfrog approach.

Q8. Producer Consumer

Producer Consumer problem is based on a producer filling in data into a memory and a consumer reading from the memory location and performing an action on it. In OpenMP to achieve this we make use of sections as it would let one thread behave as consumer and the other as producer. We need to ensure pair-wise synchronisation between producer and consumer meaning if the producer produces something i.e. fills the buffer then it should say to the consumer that you can consume it i.e. perform summation of buffer value. After filling the buffer producer uses flush construct to make its value visible to all threads i.e. not only the buffer gets updated in a local cache but it gets updated all the way to the DRAM. Then producer does an atomic write of flag to ensure it writes it completely and then flush of flag. Whereas the consumer is in spin-lock it checks whether the flag is set for it or not it does so by atomic read of flag in a temp location to ensure it reads fully. If flag is set to 1 it breaks the spin lock, does its computation and flushes it to make it visible to every thread.