Lab-6 Assignment

Vaibhav Patel - 201401222
Tanmay Patel - 201401409

Hardware details:

Architecture: x86_64
CPU op-mode(s): 32-bit, 64-bit
Byte Order: Little Endian
CPU(s): 12
On-line CPU(s) list: 0-11
Thread(s) per core: 1
Core(s) per socket: 6
Socket(s): 2
NUMA node(s): 2
Vendor ID: GenuineIntel
CPU family: 6
Model: 63
Model name: Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
Stepping: 2
CPU MHz: 1264.218
BogoMIPS: 4804.38
Virtualization: VT-x
L1d cache: 32K
L1i cache: 32K
L2 cache: 256K
L3 cache: 15360K
NUMA node0 CPU(s): 0-5
NUMA node1 CPU(s): 6-11

Remarks:

After running both the code on mpi and openmp we have come to a conclusion that for fewer number of cores Openmp is better for fewer number of cores.

But openmp is very limited to small system. We can’tput too much processor on a single chip where we have a shared memory. Because that will burn the whole system.

OpenMp	**MPI**
Spawns threads (lighter weight)	Creates processes (need more memory)
Good for fewer cores	Good for more cores
Number of cores and thus speedup is limited (in order of 10s)	Number of cores and thus speedup is large(maximum in order of 100000s)

So both are good depending on your problem size.

Question 1:

1. Context:

Brief description of the problem.: Write a parallel program to compute value of PI using trapezoid rule.And compare with serial program.

Complexity of the algorithm (serial). : For serial implementation the complexity is O(n).

Possible speedup (theoretical):

Speedup = 1 / (p / N + s) where N is no of core.

For p = 1 and s = 0 (i.e. completely parallel code) Speedup = N.

For this problem therefore speedup possible is nearly N

Parallelization strategy and problems faced in parallelization:

Using profiling information we know that the for loop which is doing actual integration is much more expensive. More than 95% of time is taken by this for loop. So we will try to parallelize it.

We are giving different start point and end point to all the cores using MPI.

This cores compute individually and give the local_sum back to the master core(core=0). Now, the master core will combine all the local_sum and further process it.

Here, Main part is that we can usd MPI_reduce() operation but here number of cores are only 16(maximum) , so there is no difference in MPI_receive(), MPI_send() and MPI_reduce().

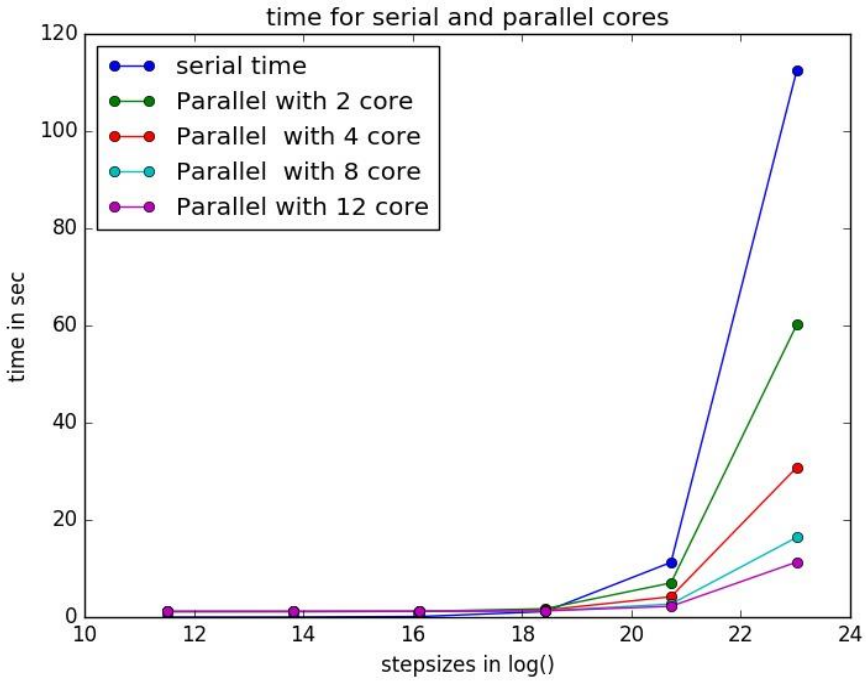
3. Parallel overhead time:

Here parallel overhead time is significant. MPI_init() is taking too much time. So if we consider time only after this line the speedup is comparable with OpenMP.

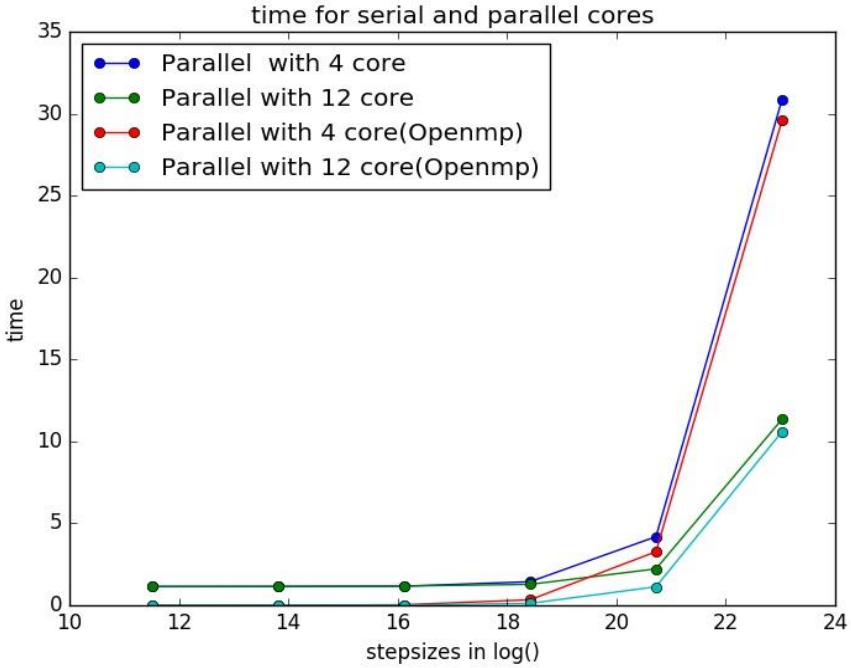
The t=MPI_init() time is increasing with number of cores. And the other time is taken by receive function. And send function.

We have put this function in clock and checked the time.

Benchmarking graphs:

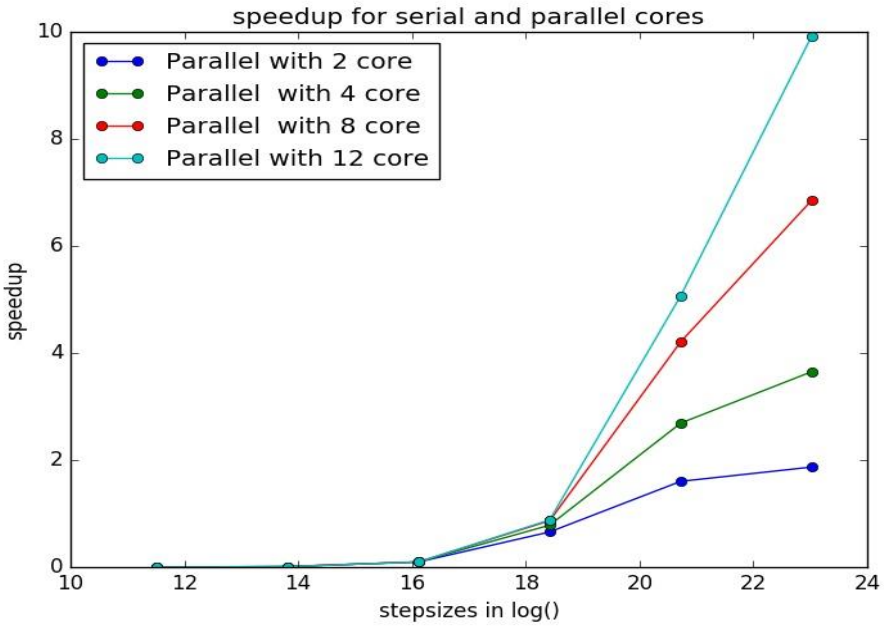


Time is nearly same for problem size less than 10^10.

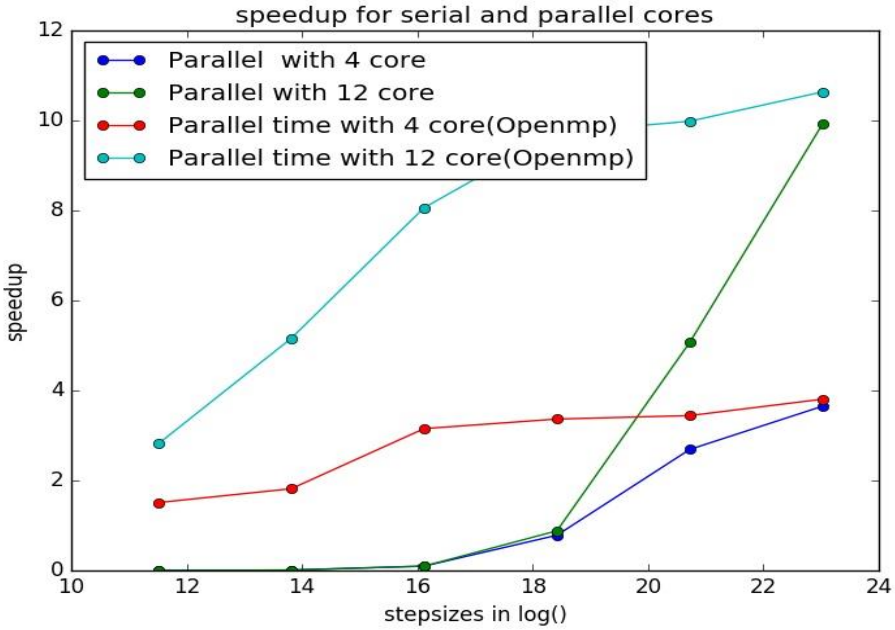


Time comparison between openmp version and mpi version.

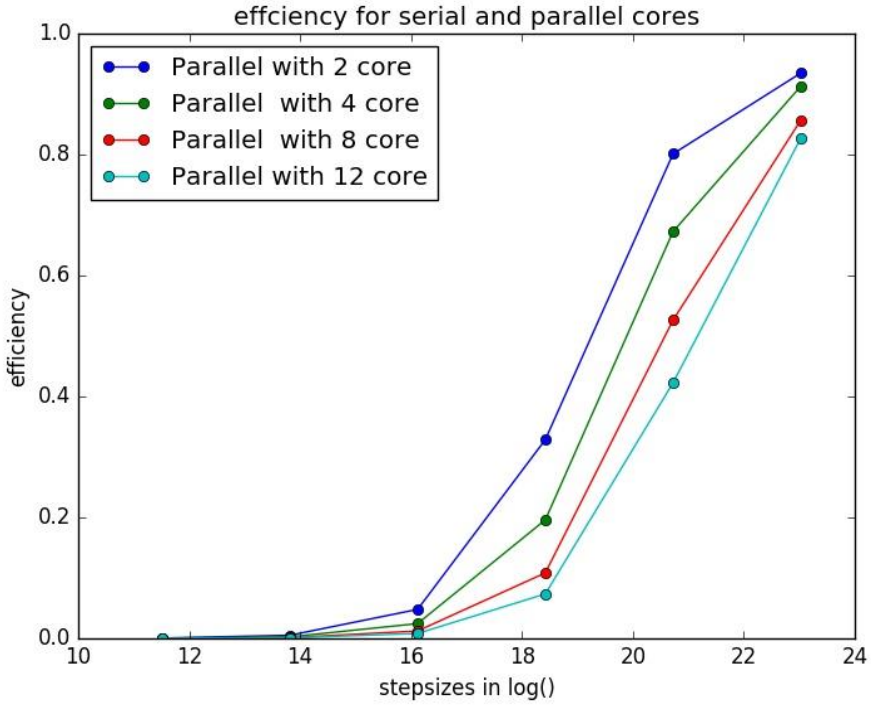
Speedup:

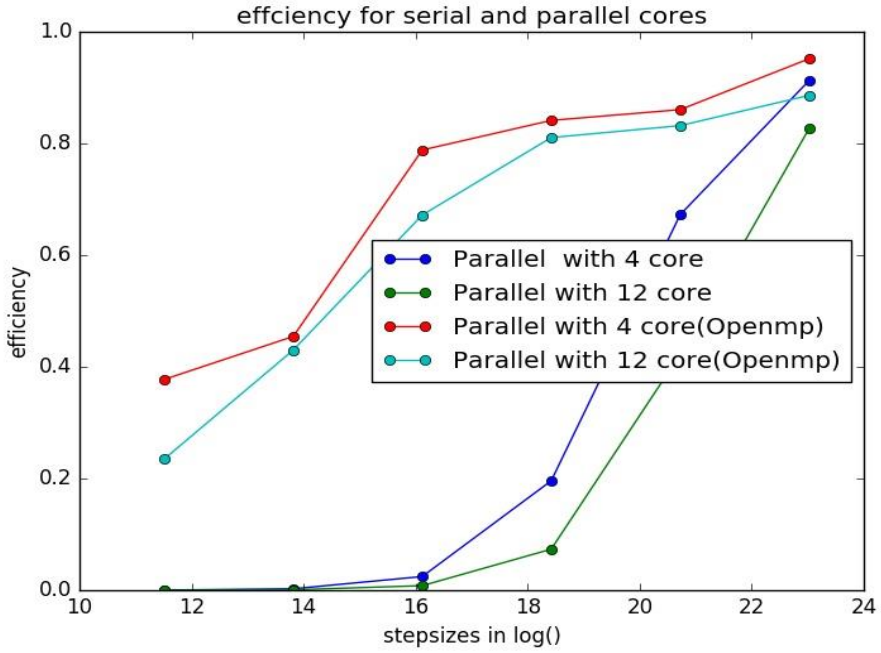


We are getting speedup of only 5 even when the problem size is as big as 10^9. This is because the actual problem time is very less compared to the overhead time(nearly 1 sec for MPI_init()).
But as we ran the code on 10^10 problem size we are getting good speedup as well as efficiency in different number of cores. There is a big change in almost all the cores (except for 2).



OpenMP is clearly winner for smaller problem size. But in 10^10 it is comparable.





Efficiency also follows the trend as the speed up is following.