High Performance Computing Report

Author 1 (Kunal Jani)
Author 2 (Pratik Ghosh)

Dhirubhai Ambani Institute of Information and Communication Technology
201601444@daiict.ac.in
201721010@daiict.ac.in

Question 1:

Implementation Details

Brief and clear description about the Serial implementation

In the serial implementation of the trapezoidal method for the calculation of the value of pi, initially taking the sum to be zero and running an iteration from zero to the total number of steps, the value of x has been incremented by 0.5 and then it is divided by the total number of steps. This value of x is used to calculate the sum of the series which has been initially set to zero. Finally the value of pi is obtained by multiplying the obtained sum by the number of steps.

Brief and clear description about the implementation of the approach (Parallelization Strategy, Mapping of computation to threads)

When MPI is used for parallel programming, threads are not created, processes are created. MPI is a distributed memory system, so the variables that have been declared before the creation of the thread will have a copy for each process that has been created. Each and every variable will behave like a private variable for each process. So for parallelizing the loop, work is distributed between the processor with each process performing an integration to calculate the value of π. The process will calculate the partial sum. When the partial sum is calculated, the partial sums are sent from all the threads to the first thread. The total of the partial sums is calculated in the first thread itself.

A better way to calculate the value of π is to invoke the reduce() function and pass the array of partial sums. The function will calculate the sum af all the partial sums and calculate the value of π.

Complexity and Analysis Related

Complexity of serial code

No of operations = n^2
Time taken = n^2
Time complexity = O(n^2)

Complexity of parallel code (split as needed into work, step, etc.)

No of operations = n^2
Time taken = n^2
Time complexity = O(n^2)

Cost of Parallel Algorithm

Cost of the algorithm = O(n^2)

Theoretical Speedup (using asymptotic analysis, etc.)

Serial execution time=p
Parallel execution time=1
$Speedup = \frac{Serialtime}{Paralleltime} = p$

Number of memory accesses

Serial algorithm:

No of memory accesses outside loop=18
No of memory accesses inside loop=$146n^2 + 4n$
Total memory accesses=$146n^2 + 4n + 18$

Parallel algorithm:

No of memory accesses outside loop=18
No of memory accesses inside loop=$146n^2 + 4n$
Total memory accesses=$146n^2 + 4n + 18$

Number of computations

Serial algorithm:

No of computations outside loop=5
No of computations inside loop=$2n + 133n^2$
Total no of computations=$133n^2 + 2n + 5$

Parallel algorithm:

No of computations outside loop=5
No of computations inside loop=$2n + 133n^2$
Total no of computations=$133n^2 + 2n + 5$

Curve Based Analysis

Time Curve related analysis (as no. of processor increases)

As the number of processors will increase, if the number of processors are small, then the execution time will be greater for a larger problem size due to the overhead cost involved in the creation of different processes.

Time Curve related analysis (as problem size increases, also for serial)

As the problem size will increase, the execution time will initially not increase much by but for larger problem sizes, for a lower number of processors the execution time will increase dramatically and for a larger number of processors, the execution time will not increase do dramatically because the division of a large amount of work among threads will be the dominant factor for a large problem size. However, it should be observed that the problem size must be larger than the case of parallel programming in OPENMP because threads are lightweight processes which are much easier to create and thus the overhead cost involved is much less compared to the case of MPI where each process is created instead of a thread.

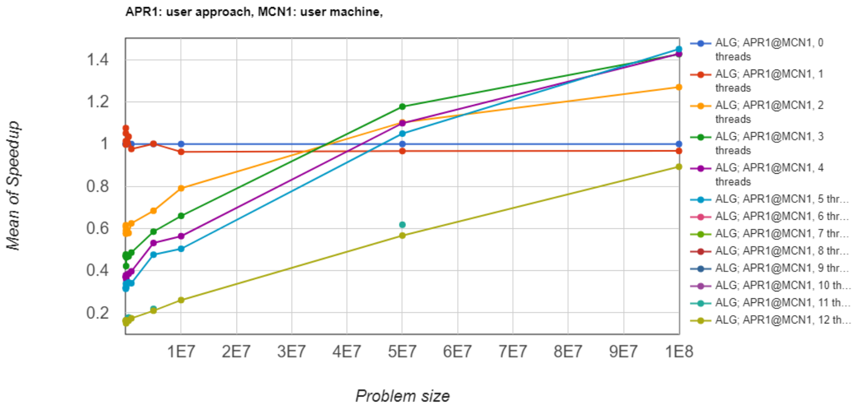
Speedup Curve related analysis (as problem size and no. of processors increase)

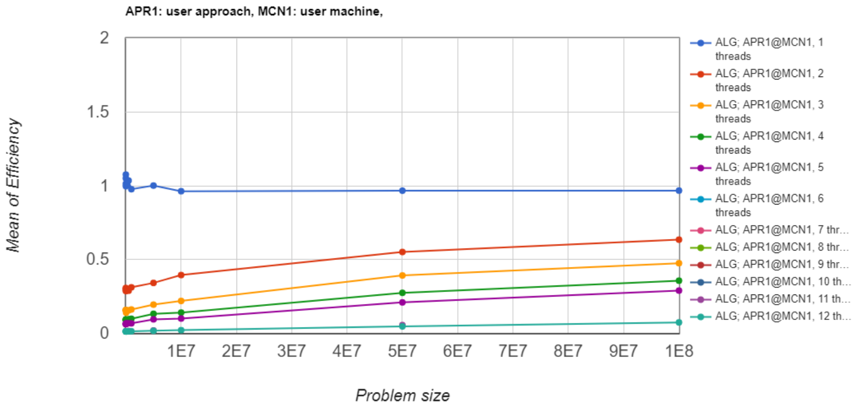
For a low problem size, the speedup will be lower than 1 because the overhead cost in the parallel algorithm dominates over the time reduces in the division of work among threads while in the case of larger problem sizes, the parallel algorithm will give a better speedup. As processes are created and not threads, the parallel execution time will be much greater thus giving a lower speedup compared to OPENMP.

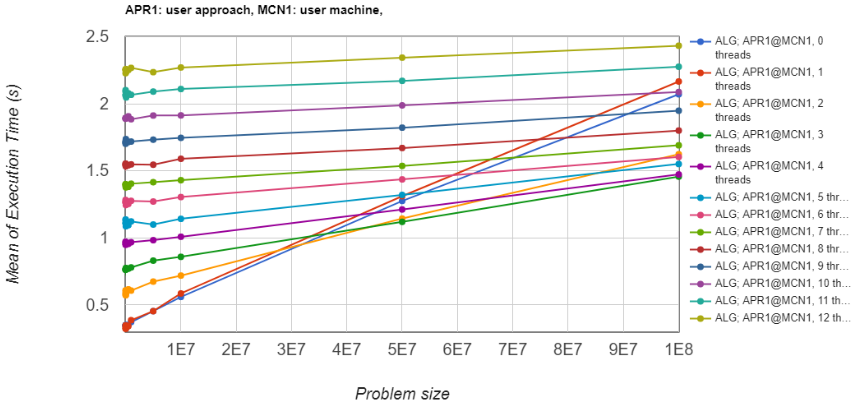
Efficiency Curve related analysis

For a lower problem size, the efficiency is much greater for the serial algorithm compared to the parallel algorithm because the speedup is much greater for a lower number of threads and the speedup to number of processors ratio is much greater in this case. For a higher problem size, efficiency will increase for the parallel algorithm for all the number of threads due to division of work among threads while it will decrease for the serial algorithm because there is no division of work. The efficiency will also reduce due to a reduction in the speedup when compared to OPENMP.

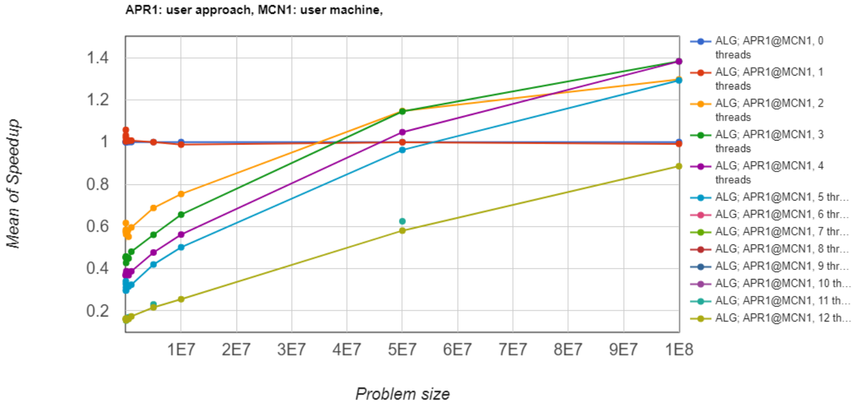
Plots for speedup, efficiency and execution time when send() and recv() are used:

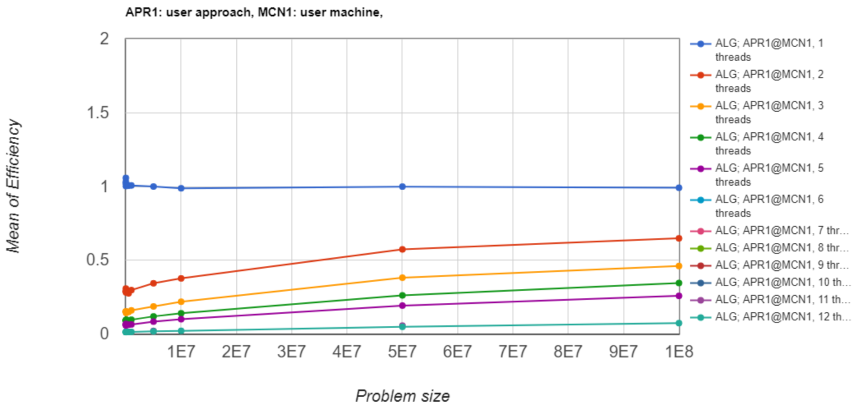


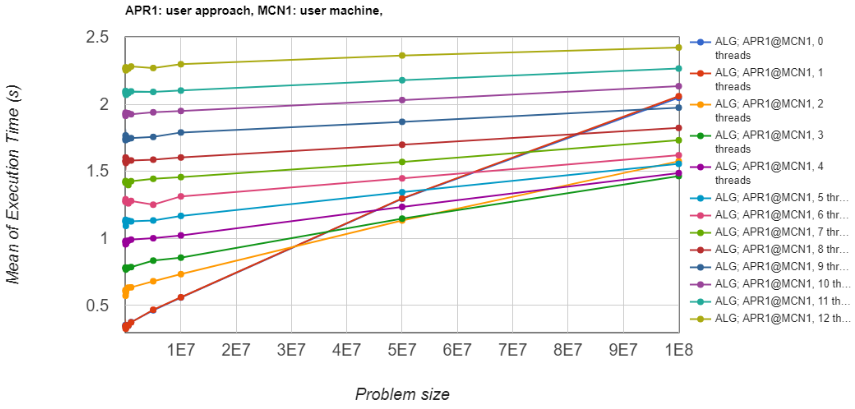




Plots for speedup, efficiency and execution time when reduce() is used:




APR1: user approach, MCN1: user machine,
Mean of Efficiency
2
1.5
1
0.5
0
1E7
2E7
3E7
4E7
5E7
6E7
7E7
8E7
9E7
1E8
Problem size
ALG; APR1@MCN1, 1 threads
ALG; APR1@MCN1, 2 threads
ALG; APR1@MCN1, 3 threads
ALG; APR1@MCN1, 4 threads
ALG; APR1@MCN1, 5 threads
ALG; APR1@MCN1, 6 threads
ALG; APR1@MCN1, 7 thr...
ALG; APR1@MCN1, 8 thr...
ALG; APR1@MCN1, 9 thr...
ALG; APR1@MCN1, 10 th...
ALG; APR1@MCN1, 11 th...
ALG; APR1@MCN1, 12 th...


APR1: user approach, MCN1: user machine,
Mean of Execution Time (s)
2.5
2
1.5
1
0.5
1E7
2E7
3E7
4E7
5E7
6E7
7E7
8E7
9E7
1E8
Problem size
ALG; APR1@MCN1, 0 threads
ALG; APR1@MCN1, 1 threads
ALG; APR1@MCN1, 2 threads
ALG; APR1@MCN1, 3 threads
ALG; APR1@MCN1, 4 threads
ALG; APR1@MCN1, 5 thr...
ALG; APR1@MCN1, 6 thr...
ALG; APR1@MCN1, 7 thr...
ALG; APR1@MCN1, 8 thr...
ALG; APR1@MCN1, 9 thr...
ALG; APR1@MCN1, 10 th...
ALG; APR1@MCN1, 11 th...
ALG; APR1@MCN1, 12 th...