CS 301
High-Performance Computing

Lab 4

Krunal Lukhi (201901449)
Dhwanil Shah (201901450)

April 20, 2022

Contents

1 Introduction

This assignment aims to compare performance of two parallelization API OpenMP ans OpenMPI for trapezoidal integration to find the value of π. We would compare performance of OpenMPI implementation with the performance of serial and OpenMP implementations in terms of speed-up, efficiency, problem size vs run time and other such performance matrices.

2 Hardware Details

- CPU - 16
- Socket - 2
- Cores per Socket - 8
- Size of L1d cache - 32K
- Size of L1i cache - 32K
- Size of L2 cache - 256K
- Size of L3 cache - 20480K

3 Part 1

3.1 Implementation description and Complexity Analysis:

- **Serial Implementation:** It involves a for loop that iterates for n steps to calculate integral. So, serial time complexity is $\mathcal{O}(N)$.
- **Parallel Implementation using OpenMP:** We have implemented three different technique of parallelization 1) naive implementation using *#pragma omp parallel* directive 2) *#pragma omp parallel for* directive and 3) using padding technique. We got the best performance in padding technique among them. Therefore, we selected it for comparison with OpenMPI implementation. Time complexity for this case is $\mathcal{O}(N/p)$ where p is number of threads. We assume that for loop will be divided equally into p threads.
- **Parallle Implemettion using openMPI:** Two parallelization technique were implemented.
  1. **Using six basic MPI calls:** The entire x interval is divided equally into p processor core. Each processor core calculates integral with given data and then sends it to master node to calculate the final answer. The communication is done using only six basic MPI calls: MPI_Init, $MPI\_Finalize$, MPI_Comm_size, MPI_Comm_rank, MPI_Send, MPI_Recv.
  2. **Using MPI_Bcast, MPI_Reduce:** The same process is carried out as mentioned in (1) using only these two MPI calls: MPI_Bcast, MPI_Reduce. T

  Time complexity in this case is $\mathcal{O}(N/p)$, where p is number of cores.

3.2 Performance analysis using graphical results

- **Run time analysis:**

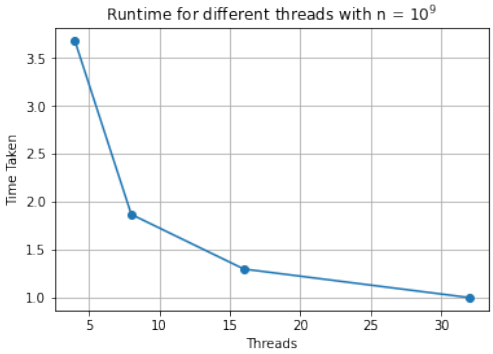

Figure 1: Question 1

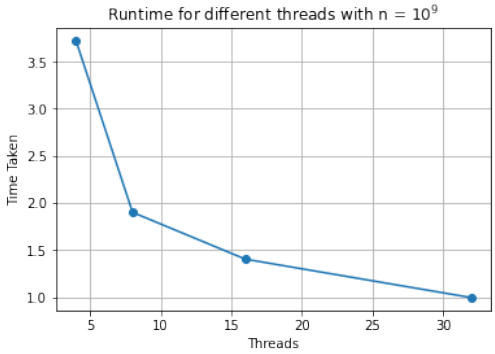

Figure 2: Question 2

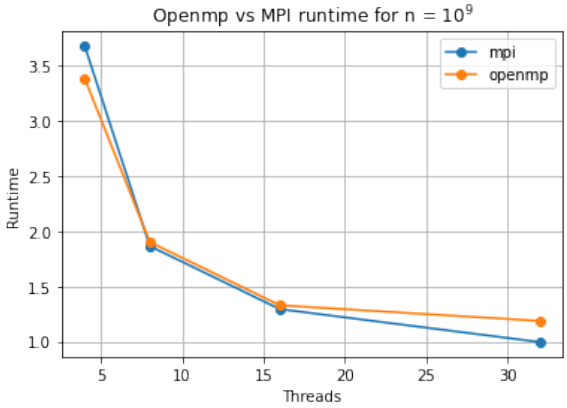

Figure 3: Comparing the runtime for OpenMP and MPI for $n = 10^9$

We observe that OpenMP and OpenMPI implementation has lesser runtime than serial implementation. However this is observed after certain number of threads/cores because for lower number of threads/cores, overhead time nullifies the performance gained through parallelization. When problem size is larger, OpenMPI implmentation perform better than OpenMP implementation as we get benefit form individual computation speed of distributed system. However, for smaller problem size, we observe that OpenMP implemntation performs better as compared OpenMPI. The reason is that overhead of OpenMPI affects the run-time significantly.

- **Speed-up analysis:**

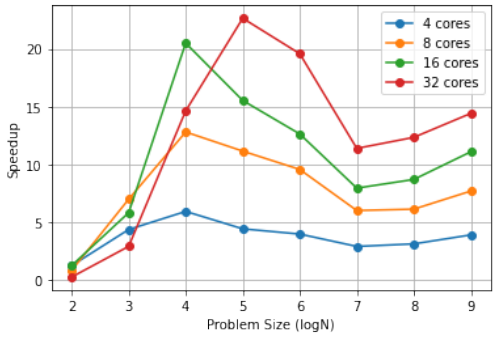

Figure 4: Question 1

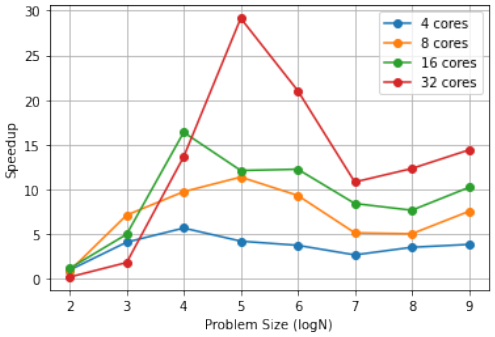

Figure 5: Question 2

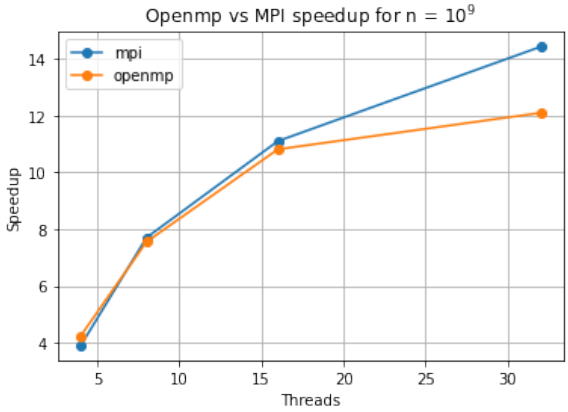

Figure 6: Comparing the speedup for OpenMP and MPI for $n = 10^9$

We can see that Question 2 achieves better speedup as compared to question 1 due to inherent nature of reduction algorithm. OpenMP has better speed up for smaller problem size while OpenMPI beats OpenMP when problem is larger. The reason is discussed in previous section.

- **Efficiency analysis:**

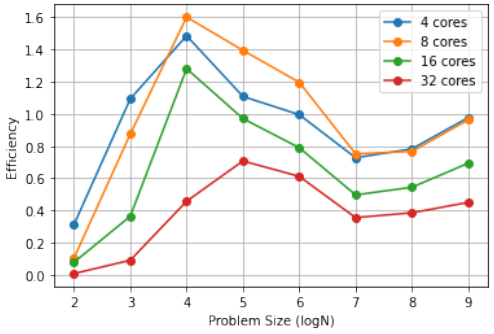

Figure 7: Question 1

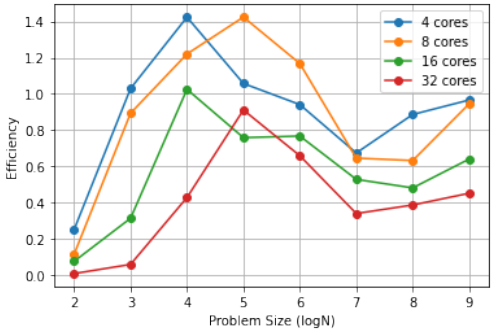

Figure 8: Question 2

Efficiency of OpenMPI is worse for smalll problem size but it improves a problem size increases. Efficiency of both OpenMP and OpenMPI decreases as number of cores/threads increases. OpenMPI is more scalable than OpenMP for larger problem size.

3.3 Obervations

1. For less number of threads/core, OpenMP performs better than OpenMPI. The reason is that communication overhead in OpenMPI affects the run time.

2. When problem size is larger, OpenMPI performs better as compared to OpenMP due efficiently parallelization by distributing work equally.

3. OpenMPI with MPI_Bcast and MPI_Reduce performs better than with basic six MPI calls.