High Performance Computing

Shantanu Tyagi, Kirtan Delwadia
201801015, 201801020

CS-301
Prof. Bhaskar Chaudhry
Due Date: 08/04/2021

Lab Date: 07/04/2021

Assignment 5

Hardware Details:

- Architecture: x86_64
- Byte Order: Little Endian
- CPU(s): 24(master), 16(slave)
- Sockets: 2
- Cores per socket: 6(master), 8(slave)
- Threads per core: 2(master), 1(slave)
- Nodes: 2
- Model name: Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
- L1d cache: 32K
- L1i cache: 32K
- L2 cache: 256K
- L3 cache: 15360K(master), 20480K(slave)
- Language: C
- Compiler: mpirun

Question 1: Trapezoidal Integration

Implementation Details:

1. Description about the Serial implementation

We numerically evaluate π using trapezoidal method using a single loop that runs for the entire length of the problem giving us the value of π at the end of the loop.

```
for(i=0;i<num_steps;i++)
{
        x=(i+0.5)*step;
        sum=sum+4.0/(1.0+x*x);
}
```

2. Description about the Parallel implementation

In parallel programming, we divide the entire problem into sub problems according to the number of cores/processors available and each processor calculates the result of the sub problem assigned to it. The processors calculate their local values of the integration and then send it to the master processor for addition. The CPU provided to us has cores divided equally among nodes. We have done 2 different implementations in MPI and compared it with the previously done calculations using OpenMP.

1. **Using the 6 basic MPI calls**
   This method is more efficient on lesser number of processors since here, the master processor sends the problem to all the processors one by one and after the processors are done calculating, they send the data to the master processor one by one thus making it less useful for larger number of processors.

```
MPI_Status status;
MPI_Init(&argc, &argv);
t_start = MPI_Wtime();
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &p);

h = ((double)(b - a) / (double)n);
local_n = n / p;
local_a = a + local_n * h * my_rank;
local_b = local_a + local_n * h;

integral = (f(local_a) + f(local_b)) / 2;
for (i = 1; i < local_n; i++) {
        double temp = local_a + i * h;
        integral = integral + f(temp);
}
integral = integral * h;

if (my_rank == 0) {
        total = integral;
        for (source = 1; source < p; source++) {
                MPI_Recv(&integral,1,MPI_DOUBLE,source,tag,MPI_COMM_WORLD,&status);
                total = total + integral;
        }
}
else {
```

```
27          MPI_Send(&integral, 1, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD);
28  }
29  if (my_rank == 0) {
30          t_end = MPI_Wtime();
31          printf("%d %lf\n", n, t_end - t_start);
32  }
33  MPI_Finalize();
```

2. **Using MPI_Bcast, MPI_Reduce**
   Here, the master processor does not send the problem data to all the processors. Rather it sends it to only a few processors and these processors then forward it to other remaining processors. This reduces the time when the number of processors is large.

```
1   MPI_Init(&argc, &argv);
2   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
3   MPI_Comm_rank(MPI_COMM_WORLD, &myid);
4   n = atoll(argv[1]);
5   start = MPI_Wtime();
6   MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
7
8   h   = 1.0 / (double) n;
9   sum = 0.0;
10  for (i = myid + 1; i <= n; i += numprocs) {
11          x = h * ((double)i - 0.5);
12          sum += 4.0 / (1.0 + x * x);
13  }
14  mypi = h * sum;
15
16  MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
17  if (myid == 0)
18  {
19          printf("%llu %.16f\n", n, MPI_Wtime() - start);
20  }
21  MPI_Finalize();
```

Complexity:

1. Complexity of serial code:

The time varies linearly with the problem size since we just have one loop which runs for n times. When the problem size is n, we have the time complexity as $O(n)$.

2. Complexity of parallel code:

Here again we have a problem with size n, however it is divided into p processors each of which runs in parallel. This means that the loop runs $\frac{n}{p}$ times on each processor giving us a time complexity of $O(\frac{n}{p})$.

3. Cost of Parallel algorithm

Speedup is calculated as ratio of time taken to execute the code serially to time taken to execute the code parallelly. The expected speed up is p but this if often not achieved due to memory access, parallel overhead, and thread synchronisation.

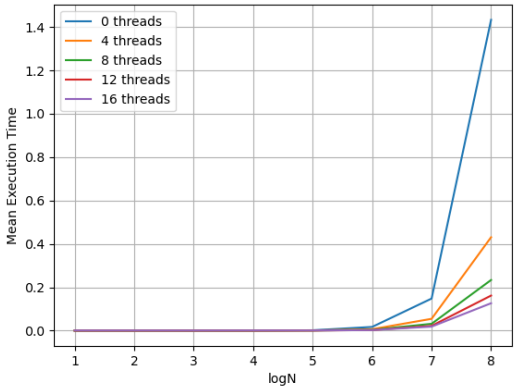
4. Theoretical Speed Up:

For a problem of size n and total p processors, we have theoretical speed up as $\frac{n}{(\frac{n}{p})} = p$
Thus for 1 processor, we get $\frac{n}{(\frac{n}{1})} = 1$
and for 4 processors, we get $\frac{n}{(\frac{n}{4})} = 4$ Speedup is calculated as ratio of time taken to execute the code serially to time taken to execute the code parallelly. The expected speed up is p but this if often not achieved due to memory access, parallel overhead, and thread synchronisation.

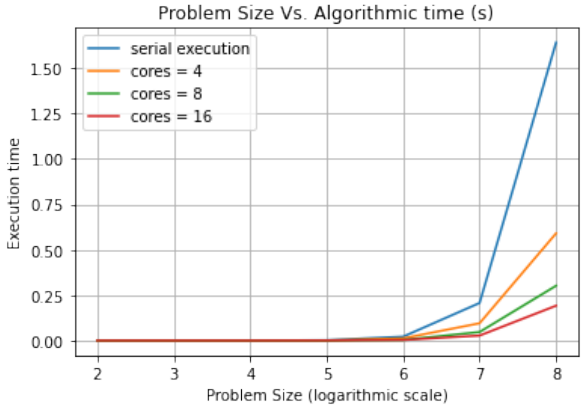
Curve Related Analysis:

1. Time Curve Related analysis:

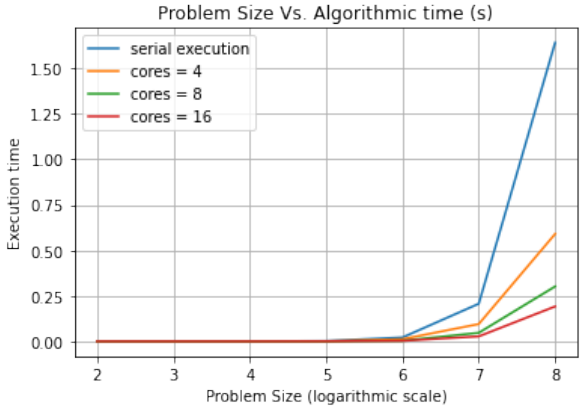
As the problem size increases, the time taken also increases for both serial and parallel implementations. This is because with increased problem size, more calculations need to be done and the loop runs longer. For a given problem size, as the number of cores increase, the execution time decreases since the work gets divided between the cores. The second implementation gives slightly better results for larger problem size because of the reasons mentioned earlier. However, the OpenMP implementation gave lesser execution times compared to the MPI implementations for all problem sizes and also for different cores taken.



(a) Atomic



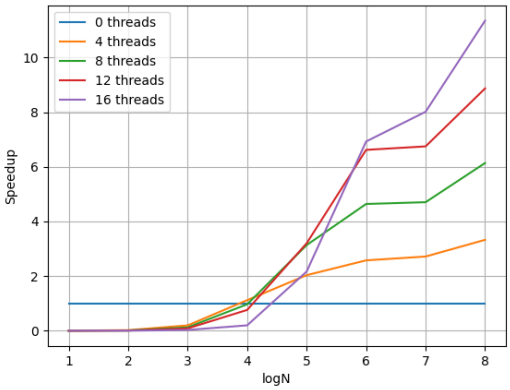
(b) Implementation 1



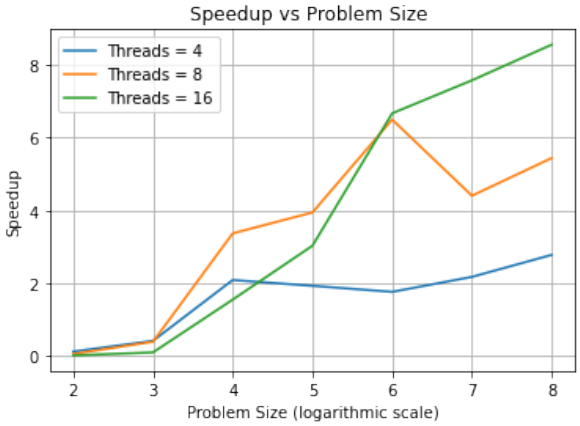
(c) Implementation 2

2. Speedup Curve Related analysis:

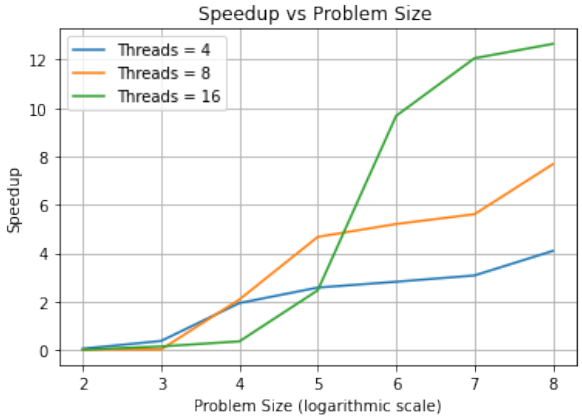
The speed up increases as the number of processes increases on which the code runs. However, if we increase the threads beyond a certain threshold than the cost to create and synchronize threads will become significant giving poorer results. As the problem size increases, the speed up reaches closer to number of processors involved. The deviation in these values from the theoretical values depicts that for large problem size the memory access time also comes into consideration. For larger problem sizes, implementation 2 gives better results than the other. OpenMP performs better than the first MPI implementation and worse then the second MPI implementation.



(d) Atomic



(e) Implementation 1

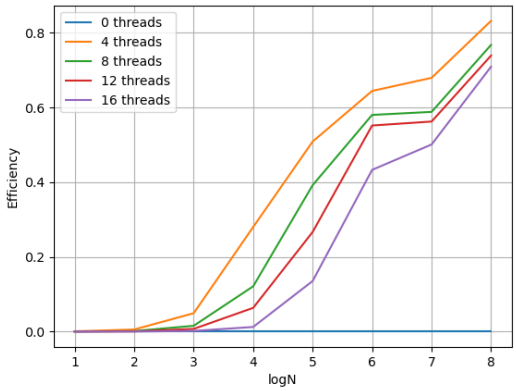


(f) Implementation 2

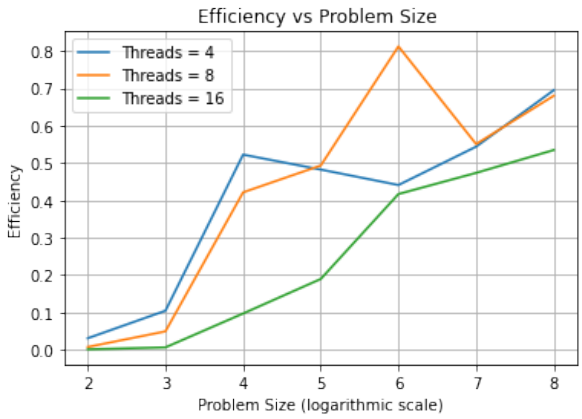
3. Efficiency Curve Related analysis:

Efficiency is speedup divided by the number of threads, p. It is also observed that with greater number of processors working together the efficiency also decreases. The greater number of threads lesser will be the efficiency because managing so many threads also requires time and the parallel overhead starts dominating. Cost of creating more and more threads is not able to balance how much more performance it gives as compared to lesser number of threads. As the problem size increases the efficiency also gradually starts increasing for a given implementation and gradually reaches 1. For larger problem sizes, implementation 2 gives better results than the other. For larger problem sizes, implementation 2 gives better results than the other.

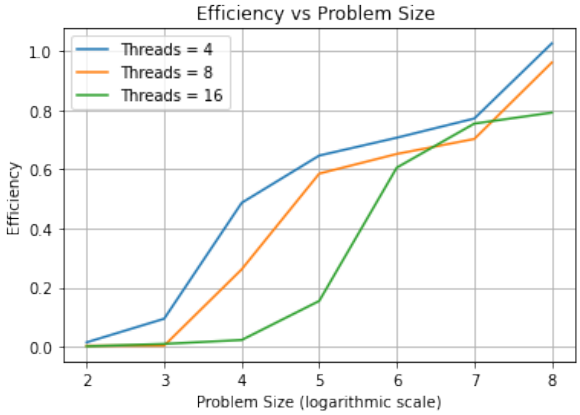
OpenMP performs better than the first MPI implementation and worse then the second MPI implementation.



(g) Atomic



(h) Implementation 1



(i) Implementation 2

Figure 1: Efficiency

4. Conclusion:

OpenMP is a shared memory device program where as MPI is a distributed memory device program, which means that the overheads in OpenMP are less compared to MPI. The communication between nodes is difficult and expensive compared to communication between shared memory systems.