CS410: Parallel Computing

Assignment 3

1. Introduction

In this assignment, we implemented a parallel version of the SUMMA matrix multiplication algorithm using the Message passing paradigm and conducted various experiments related to parallelism, whose observations can be seen in this report.

2. Implementation

Let p denote the number of MPI processors, and the size of the matrices be n x n.

MPI_SUMMA(comm_cart, b, b, b, A_loc, B_loc, C_loc):

```
1. Determine the cartesian coordinates of the processor in the
   communicator. Let this be row_coor and col_coor.
2. Partition the cartesian topology comm_cart using
   MPI_Cart_sub for rows of A. Let this be called row_comm.
3. Partition the cartesian topology comm_cart using
   MPI_Cart_sub for columns of B. Let this be called col_comm.
4. Create a local copy of A_loc, B_loc, C_loc.
5. for bcast_root=0...n/b:
6.   root_col = bcast_root
7.   root_row = bcast_root
8.   if col_coor == root_col:
9.        Copy the local copy of A_loc saved earlier to A_loc
10.  Broadcast A_loc from root_col within row_comm
11.  if row_coor == root_row:
12.       Copy the local copy of B_loc saved earlier to B_loc
13.  Broadcast B_loc from root_col within col_comm
14.  Using naive matrix multiplication multiply A_loc and B_loc
   and store the result in C_loc_temp
15.   C_loc = C_loc + C_loc_temp
```

```
1. Create a 2D cartesian communicator with sqrt(p) number of
   processors along each dimension. Let us call this comm_cart.
```

2. Assign each processor its local copy of the matrices, i.e. assign it a block of a matrix of size n/sqrt(p) x n/sqrt(p). Let these blocks be called A_loc, B_loc, and C_loc.
3. b <- n/sqrt(p)
4. MPI_SUMMA(comm_cart, b, b, b, A_loc, B_loc, C_loc)

In the above pseudo-code, we divide the work between p processors by dividing the matrix into sqrt(p) x sqrt(p) blocks. Each processor owns a particular block of the matrix. Now, we broadcast the block of matrix A and B to all the processors that have the same column and row as the root processor, respectively. Once the block of the matrix is received, we perform the matrix multiplication serially and add the result to C.

3. Experiments

Processors	Run 1	Run 2	Run 3	Average	Speedup	Efficiency
1	221.3227	223.122	225.8035	223.4161	1.0	1.0
4	55.38476	52.95572	53.40985	53.91678	4.143721	1.03593
9	25.10135	24.39314	24.18643	24.56031	9.096634	1.010737
16	16.60356	16.83629	15.5433	16.32772	13.68324	0.855203
25	11.66874	11.26458	11.26433	11.39922	19.59925	0.78397
36	10.00572	10.22239	10.5105	10.24621	21.80476	0.605688
49	8.53636	9.006113	8.516567	8.686347	25.72038	0.524906
64	8.577056	8.140449	8.64987	8.455792	26.42167	0.412839

Note that the above results are obtained for n=2 nodes. The execution time for one processor is the time for serial execution.

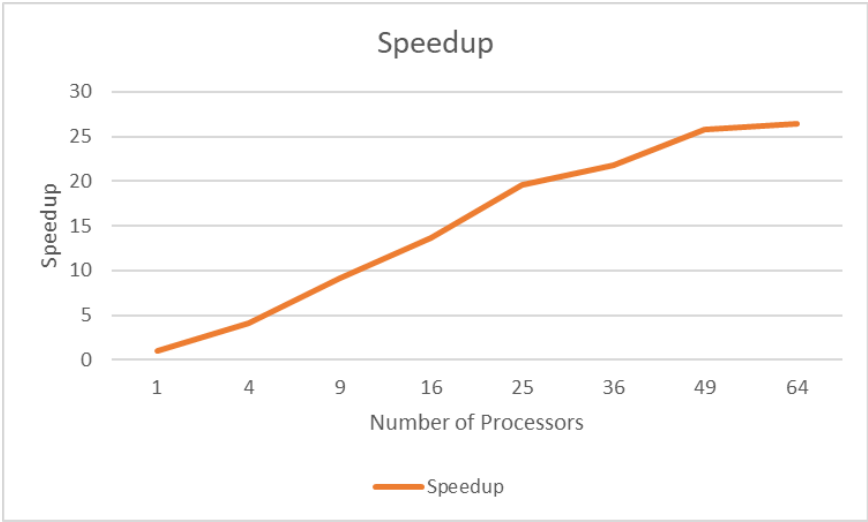

Fig. 1: Speedup Vs No. of processors

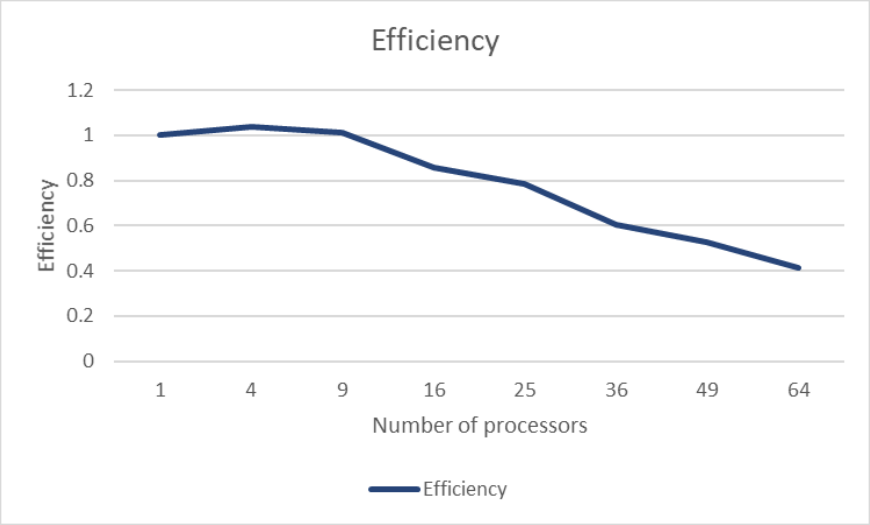

Fig. 2: Efficiency Vs No. of processors

The trends for speedup and efficiency are as expected. This is because as we increase the number of processors, the execution time decreases. Initially, the decrease in the execution time is significant thereby we see a huge bump in the speedup. However, once the diminishing returns of increasing the number of processors peeks in, the reduction in the execution time is not that great, and the speedup graph plateaus.

We observe a decreasing trend in efficiency as the number of processors increases. This is because as we increase the number of processors, the execution time decreases but at the same time, the overhead caused by the parallelism also increases. Initially, both of the opposing forces cancel each other hence we observe efficiency values near 1. However, as the number of processors is increased further, the overhead is so much that the efficiency decreases.