Reah Rajmangal
NETID: rr2886

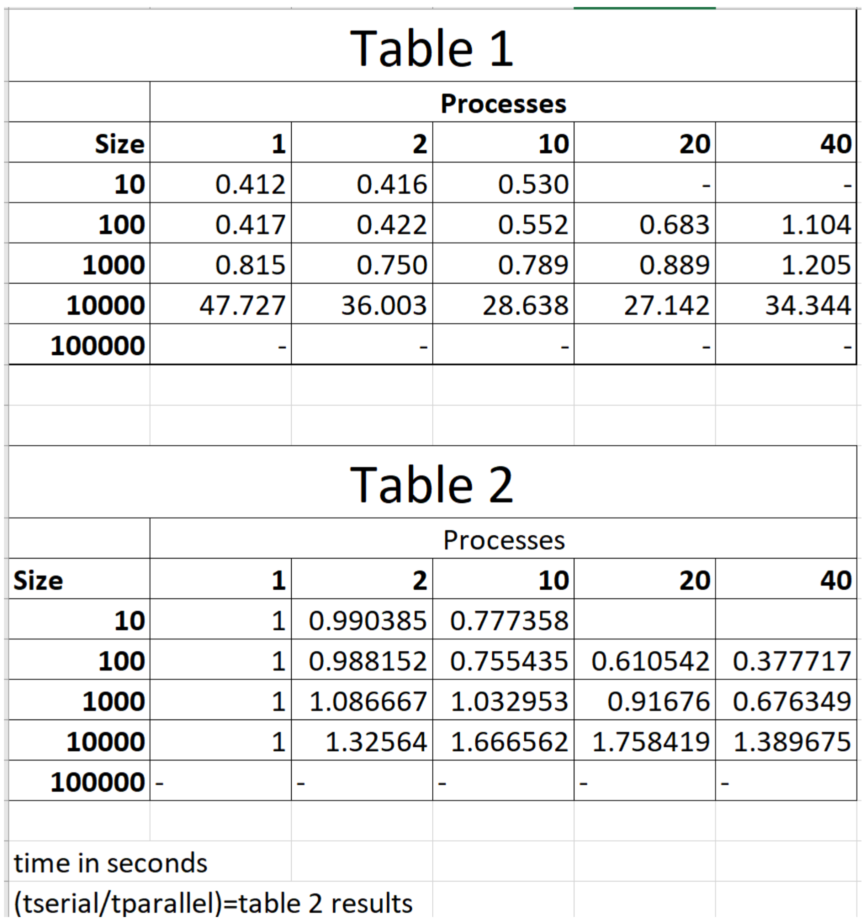

Table 1

	Processes				
Size	**1**	**2**	**10**	**20**	**40**
10	0.412	0.416	0.530	-	-
100	0.417	0.422	0.552	0.683	1.104
1000	0.815	0.750	0.789	0.889	1.205
10000	47.727	36.003	28.638	27.142	34.344
100000	-	-	-	-	-

Table 2

	Processes				
Size	**1**	**2**	**10**	**20**	**40**
10	1	0.990385	0.777358		
100	1	0.988152	0.755435	0.610542	0.377717
1000	1	1.086667	1.032953	0.91676	0.676349
10000	1	1.32564	1.666562	1.758419	1.389675
100000	-	-	-	-	-

time in seconds
(tserial/tparallel)=table 2 results

Was not able to do 100000 due to no space in memory on CIMS server…

A. There is a more significant slowdown (~0.75 speedup) when problem size is 10 and number of processes is 10, problem size 100 and number of processes is 10. as well as almost a half reduction when problem size is 100 and number of processes is 20 and 40 as compared to the sequential run of the program, along with problem size 1000 and number of processes 40. We see a small amount of slowdown (~0.90+ speedup) as well when there are 2 processes and 10 or 100 problems and 1000 problems with 20 processes. In general, there is not a speedup when the problem size is small (10/100) and the process size increases. In fact, speedup reduces as the number of processes increases when problem size remains the same.

B. There is no speedup in the case of a small number of problems handled with multiple processes due to the communication among processes which can lead to an increase in time to handle these communication and synchronization such that it can take more time than sequentially for one process. For example, in my code, I am calling an allgather function which is a collective call that syncs up the processes so that I can get a concatenated array of new Xs as well as another allgather function to collect the boolean value (0,1) for each process and iterate through this array to see if it is necessary to continue iteration. The time it takes for these communications to sync the processes can take more time than solely computing the new X's, updating the original x array and checking the errors of all new X's altogether. Hence it does not make sense to do this especially for smaller number of processes.

In bigger cases, such as problem size 1000, when dealing with 20 or 40 processes, there was a reduction also likely due to the overhead time it takes for processes to communicate and gather the necessary information to check if we should do another iteration.

C. There is barely a speedup (~1.00+) when process size is 2 or 10 and problem size is 1000. There is more significant speedup (1.30+) when the problem size is very large (10000). In fact, speedup increases as the number of processes increase for problem size 10000 and the speedup for 100,000 problem size would also likely have more speedup when we increase number of processes.

D. There is speedup in these cases, especially when problem size is 1000, because we spend less time doing computations in parallel than it would sequentially (for example, 1000 for 10 processes, 100C per process as opposed to 1 process handling 1000C computations, C being a constant number of comps per process). The processes, though spending time communicating, are spending a significantly less amount of time computing new Xs per process.

Since each process has almost, if not same, amount of computations, their communication cost to sync within the while loop is very low, the load imbalance would be close to 0 if not 0 already due to the equal amount of calculations. It is interesting to note that speedup is highest when 10,000 equations are split between 20 processes (as opposed to 40 processes). This could mean that ~20 processes are an ample amount to calculate 10,000 problems given this algorithm and raises an important point that even though the number of problems to solve are high, computing in parallel will only increase performance to a certain extent (number of processes), after which no additional (or perhaps decrease) performance will be shown.