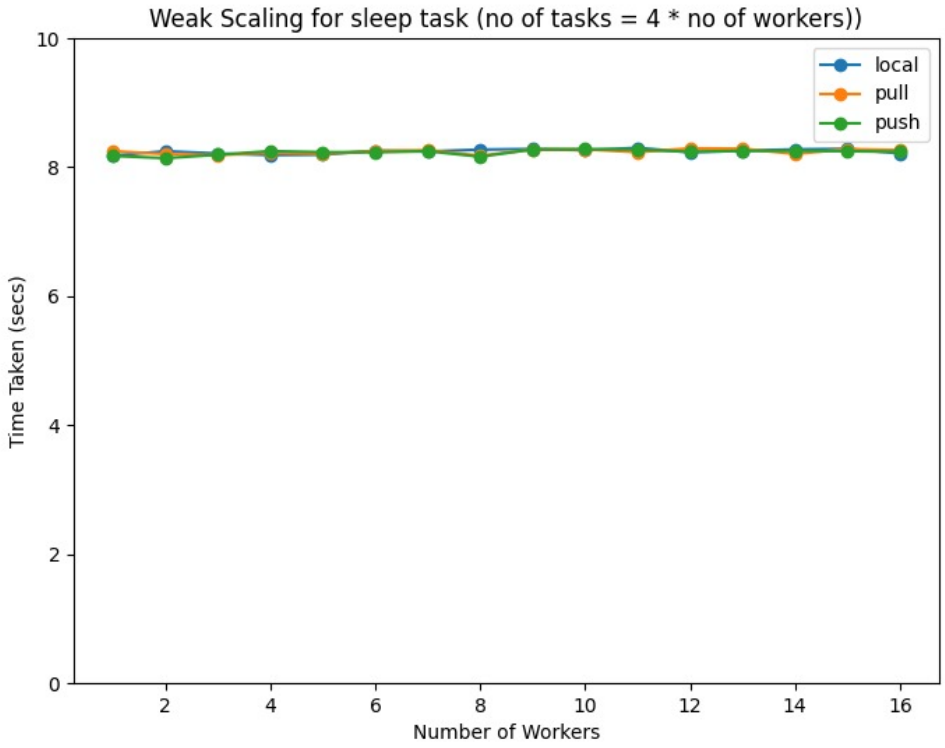
Performance Report

Weak Scaling

Weak scaling is a concept used in parallel computing to measure the efficiency of a parallel algorithm as the problem size and the number of computing resources increase proportionally. In weak scaling, the workload per processor remains constant as more processors are added to solve a larger problem.

Weak Scaling on sleep task
The sleep task executes a function that just sets the thread to sleep for given number of seconds. So, *time.sleep(t)* is the only code being run in the function.

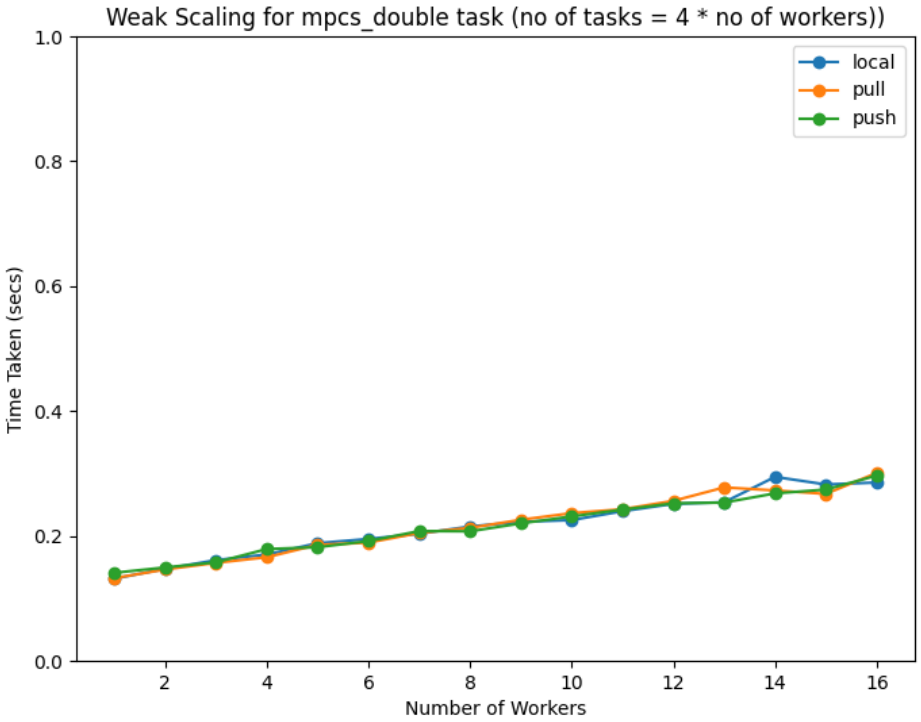


The plot above is developed for sleep time = 2. So, an execution time of around 8 secs was expected since the number of tasks = 4 * number of workers.
The plot shows perfect results as far as weak scaling is concerned. Although, since the number of physical cores of the system the program is being run are 8, we get a perfect result for number of workers > 8. This is because the task is not

compute intensive and it just gives up processor as soon as it is run since sleep is the only statement it executes. Also, it is noteworthy that the tasks are distributed evenly among all the workers since the tasks are long.

Weak scaling on mpcs_double task

mpcs_double task involves just returning a value which is twice the input parameter.

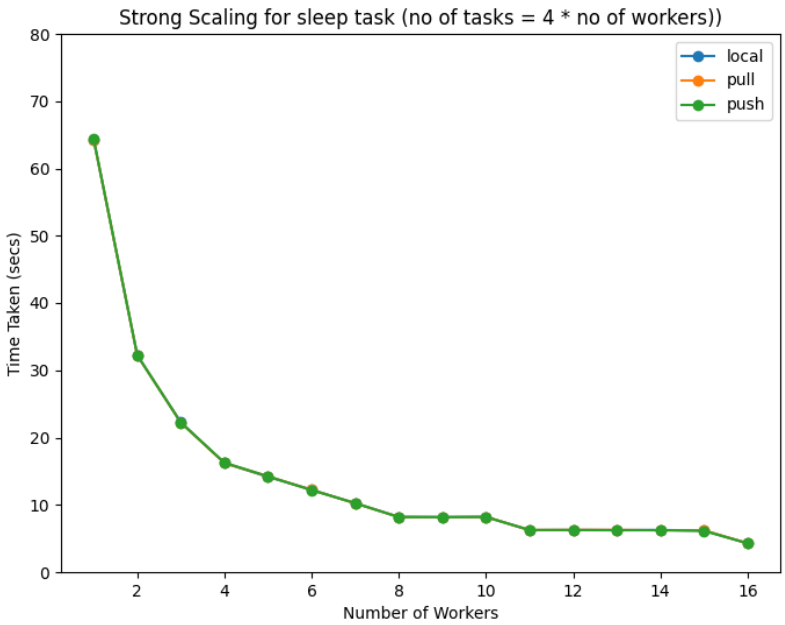


In this case, the task is so small that the worker is freed immediately after being assigned the task. So, by the time a new task is in the process of being assigned, the first task is already complete, essentially making the entire process serial. Hence, a linear scaling of time taken is expected for this task.

Also, the Gustafson's law, is a principle related to weak scaling that suggests that the speedup achieved by adding more processors to a parallel system can be significant if the problem size also increases. So, as the number of processors increases, the execution time can be reduced by solving larger problems. This law is directly applicable for the present scenario.

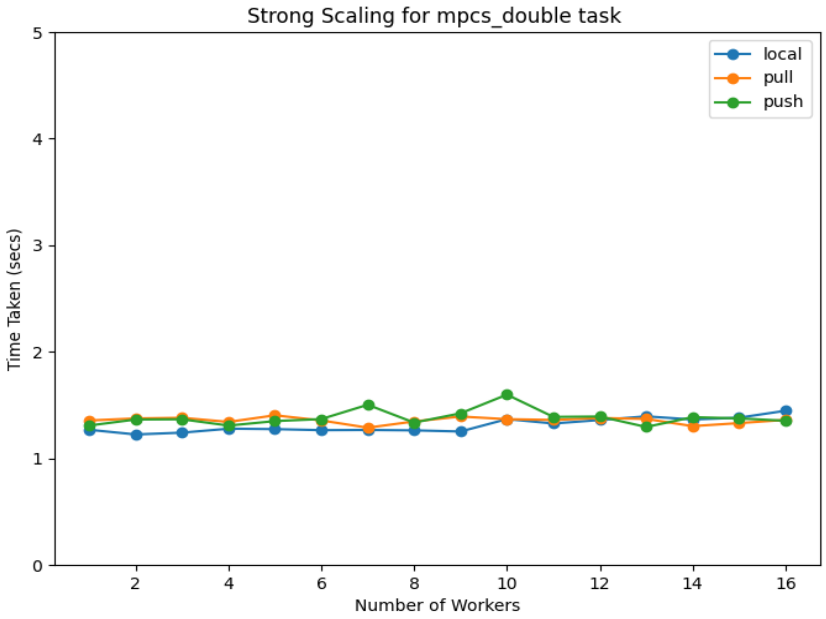
Strong Scaling

Strong Scaling on sleep task



For this case, we get perfect scaling, precisely because of the same reason mentioned above for weak scaling.

Strong scaling on mpcs_double task



In this case, again since the tasks are very small and since strong scaling involves using a constant number of tasks for varying number of workers, the time taken remains the same. Each task completes immediately and makes the processor available for the next task almost instantaneously. So, this essentially creates no difference when we try to run the code with varying number of workers, resulting in the above plot.

Latency

Latency is basically the time taken to run a single task from beginning till the end with a single worker. We compute latency for all three modes :

1. Push : 1.583 sec
2. Pull : 1.596 sec
3. Local : 1.386 sec

We can observe that the latency overhead for local worker is minimum, and the pull and push workers have a larger overhead due to ZMQ communication over network and polling.

Drawback and Possible Improvements:

1. In the no-op/ double function task case we observed that the task is so short lived that the execution terminated by the time the next task is assigned. Here the communication overhead is much larger than the time taken to complete the task. We can improve upon this by instead sending small workloads in chunks rather than one at a time.
2. At a few places we are required to use non-blocking recv sockets which are polling continuously. This causes wastage of clock cycles and also requires setting an arbitrary empirical polling interval duration. This could be improved by instead using something on the lines of a condition variable, where the thread yields when there is no data and is woken up only when there is data available
3. REP/REQ repeated disconnects since communication is only over 1 port. We could make it better by using some DEALER/ROUTER version where the connnection can be persisted over the life time of the workers.