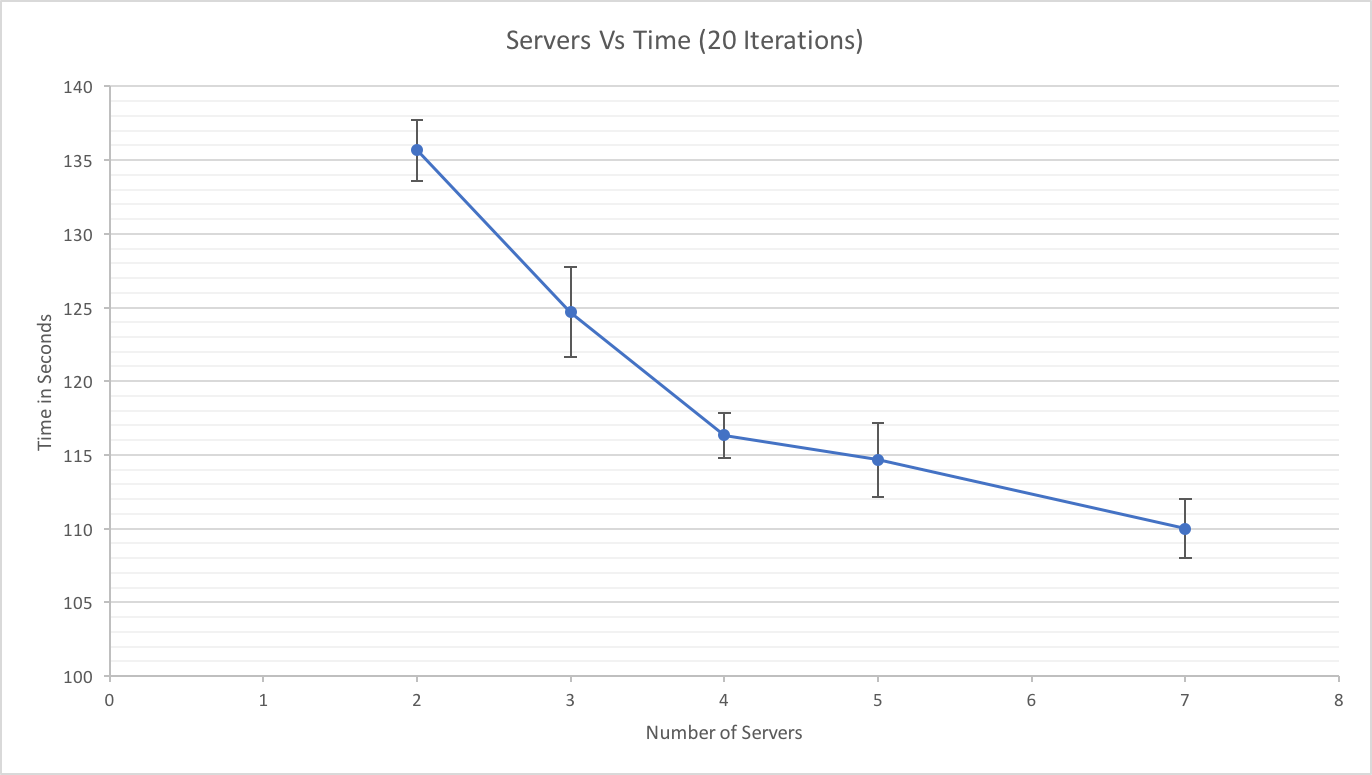
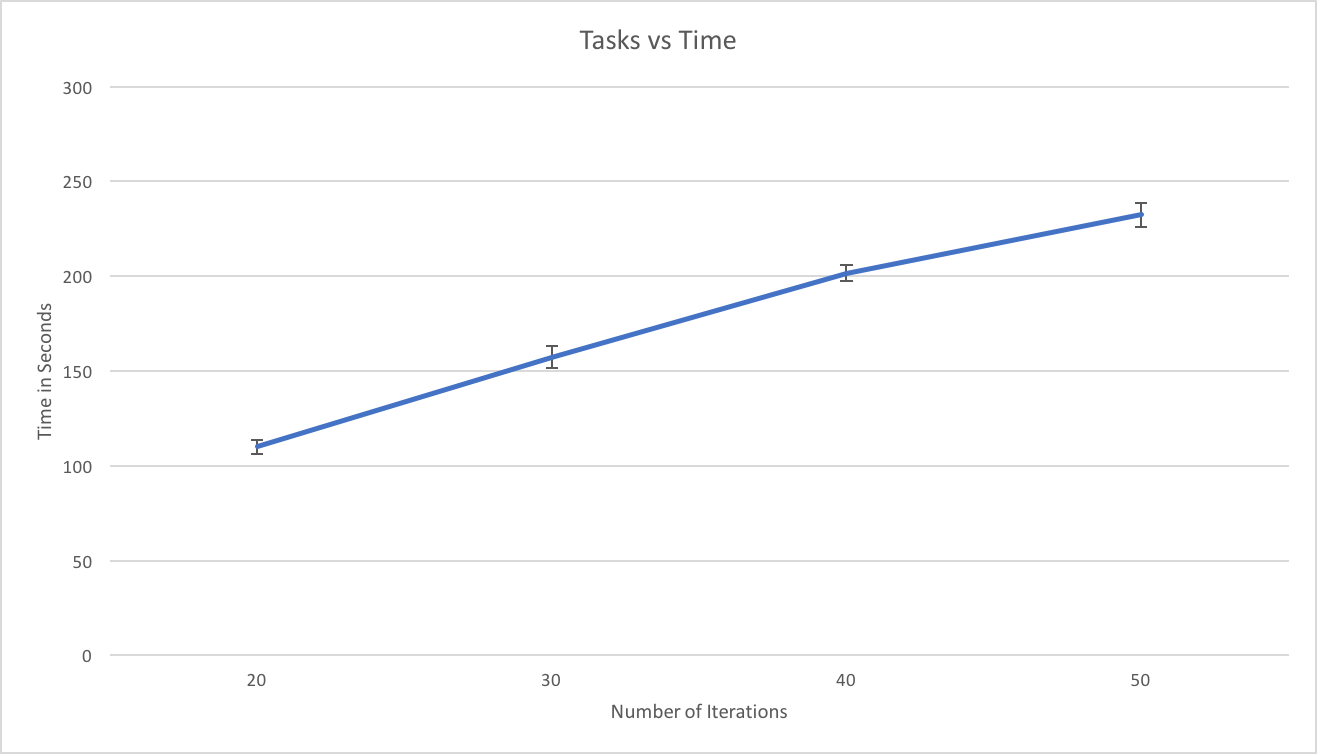
**Design and Architecture:**

In our SAVA design, we have a Master, a Client and a hot standby Master. The Client sends out the input graph to the master. The master partitions the graph and sends partitioned graph files to the workers for them to load the vertices in their respective data structures. We account for the undirected nature of the graph by initializing the ending vertices as well. We use md5 hashing to aid with partitioning and NUM\_PARTITIONS variable is used to determine the number of partitions per worker. A state class holds information about the vertex including incoming message queues and the queues for the S+1 super step and the list of outgoing edges. Additionally, the client sends out the compiled application.so file which gets loaded as a dynamic library in each worker. We then create an object of each active vertex per iteration that executes the program in each super step. An Active flag per worker is set if there is at least one worker active for the next iteration. This helps with the barrier synchronization step wherein we make sure that we have gotten acknowledgements from each of the previously active workers at the end of the super step so that the master can start the next super step. We used C++ for building SAVA and dynamic loading of libraries to provide the client the flexibility of sending a pre-compiled piece of code for graph processing task. If the Master fails, the standby master takes over. It's a hot standby so everything that the Master has, the standby receives as well. This way master failure is handled. The Client is VM1 which is also the introducer. It doesn't fail and this makes sure that any machine can join and leave at any time. If worker fails, the task is restarted. The design is very like Pregel and adopts all the crucial design decisions. When all workers are inactive, the algorithm halts and the job is completed.



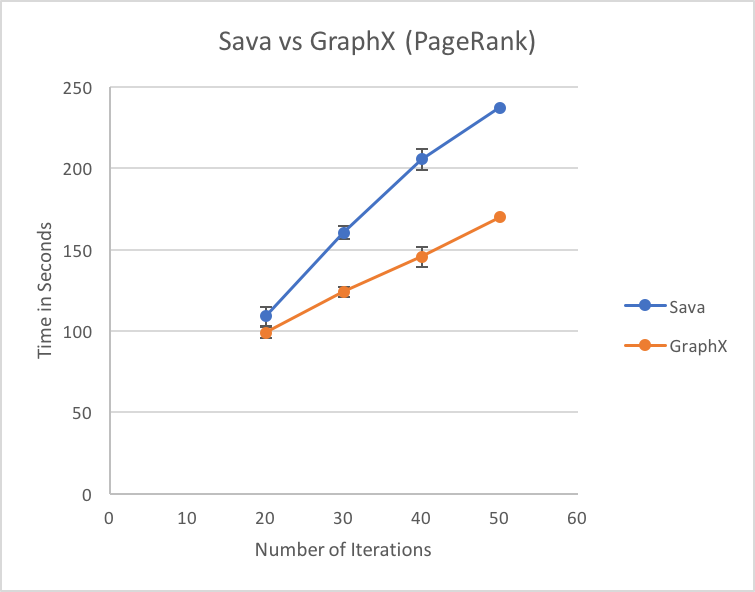


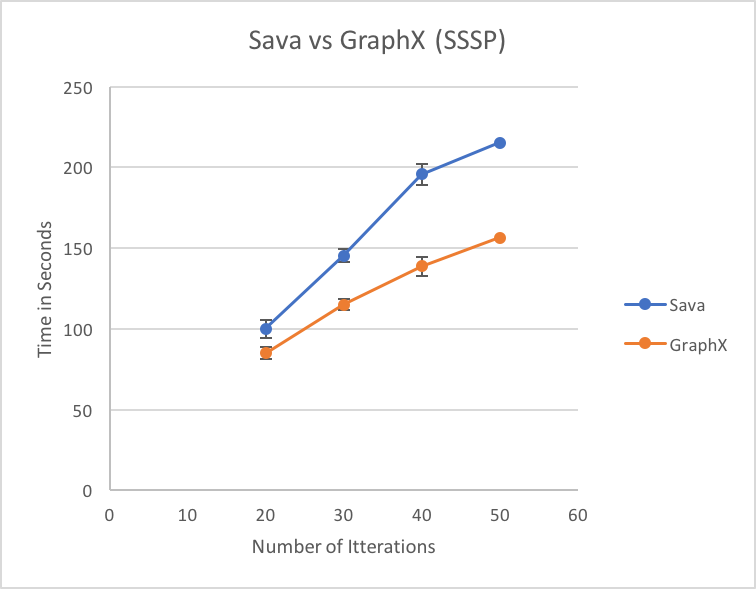
**For tasks vs time:**

We observe that as the number of iterations increases, the overall job runtime increases as well. Each iteration is dependent on the previous values and each iteration takes almost the same time to process the graph. Hence the graph is almost linear and time taken increases with number of iterations.

**For Servers vs time:**

We observe that as the number of servers increases, the time taken for processing the graph decreases. This is because we now have a lot of servers working in parallel to produce the results with the same number of iterations. The graph gets divided into more partitions and individual threads handle these partitions.





The above graphs are time vs number of iterations.

We observe that SAVA takes more time than GraphX. This is because we did not implement a more efficient partitioning function and thus partitioning the input graph, followed by hashing the vertices to the right partition, creating the Sharded files, forwarding these to all alive vertices and loading the vertices on the workers into their data structures - all have their overhead.