Performance Report: Distributed FasS System

Overview:

This performance report presents the results of a weak scaling study conducted on our distributed FaaS system, evaluating its performance across different tasks and varying the number of workers and types of tasks. The study involves four distinct tasks: Latency/concurrent requests (no-op), Floating Point Math (Python), Floating Point Math (Numpy), and Non-floating point math. The key metric measured is the average time per task, with the aim of assessing scalability.

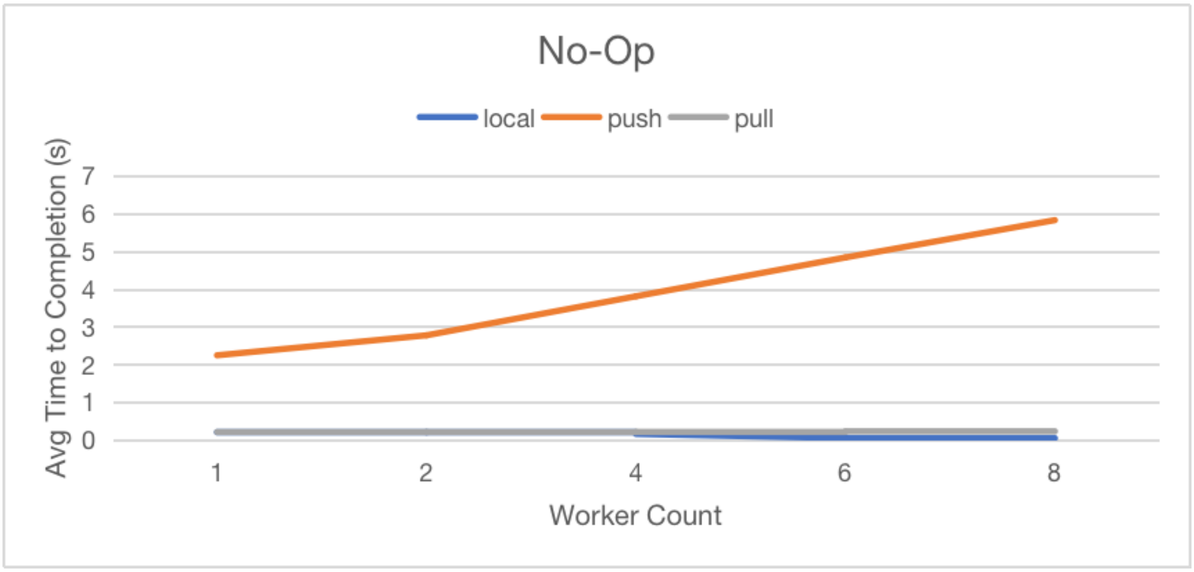
Experimental Setup:

1. Task Configuration: Fixed quantity of tasks per worker (50 tasks per worker for no-op tasks, 5 tasks per worker for load tasks).

2. Function Registration: The Pass function in Python for no-op tasks, matrix multiplication written in Python for Floating Point Math, matrix multiplication from the Numpy library for Numpy tasks, and a spin function for Non-floating point math.

Table 1: Latency/concurrent requests (No-op)

workers	local	push	pull
1	0.2091	2.245	0.2123
2	0.2093	2.7734	0.2127
4	0.1636	3.8064	0.2155
6	0.05	4.8345	0.2326
8	0.045	5.8222	0.2455



Latency/concurrent requests (No-op):

Local Approach:

1. Shows positive weak scaling, with a decrease in average time per task as the number of workers increases.
2. It likely benefits from local processing and minimal communication overhead.

Push Approach:

1. Exhibits an increasing trend in average time per task with more workers.
2. Indicates potential inefficiencies in the push method or increased communication overhead.

Pull Approach:

1. Demonstrates positive scaling, similar to the local approach.
2. The pull approach might be more efficient in handling no-op tasks compared to the push approach.

Analysis:

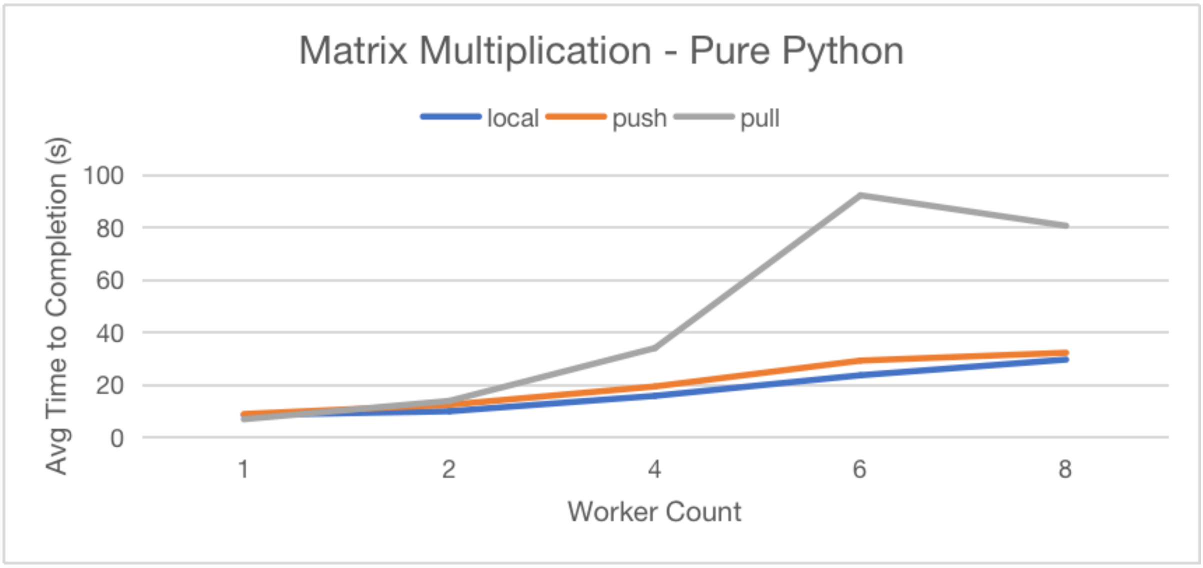
1. By utilizing the pass function, which incurs a minimal computational cost, one can derive insights into the latency of concurrent requests. The significant disparity in costs among the three approaches—local, pull, and push—is attributed to the resource-intensive nature of the push worker. This is primarily due to the task dispatcher's necessity to await registration messages from push workers. Moreover, the associated costs stem from the verification of worker liveness and the prevention of task loss through the recording of task IDs. Consequently, the setup time for the push worker can considerably exceed the actual task processing time.
2. The reason behind the local worker incurring a minimum cost lies in its independence from communication with other nodes. As it operates exclusively within a local

context, there is no need for inter-node data exchange or coordination, resulting in a more streamlined and cost-efficient process.

3. Similarly, the Pull worker's effectiveness can be attributed to its utilization of the REQ/REP (Request/Reply) pattern. This communication pattern ensures a reliable and synchronous interaction between the worker and the system. By employing a straightforward request and response mechanism, the Pull worker efficiently manages communication, contributing to its overall effectiveness in handling tasks. The clarity and predictability of the REQ/REP pattern help minimize overhead and enhance the worker's performance.

Table 2: Floating Point Math (Python)

workers	local	push	pull
1	8.3472	8.7455	6.7635
2	9.8085	12.2908	13.7607
4	15.6808	19.2678	33.9221
6	23.5469	29.1358	92.1998
8	29.5167	32.1351	80.5819



Floating Point Math (Python):

All Approaches:

1. Poor scalability was observed across all methods, with an increase in average time per task as the number of workers increased.

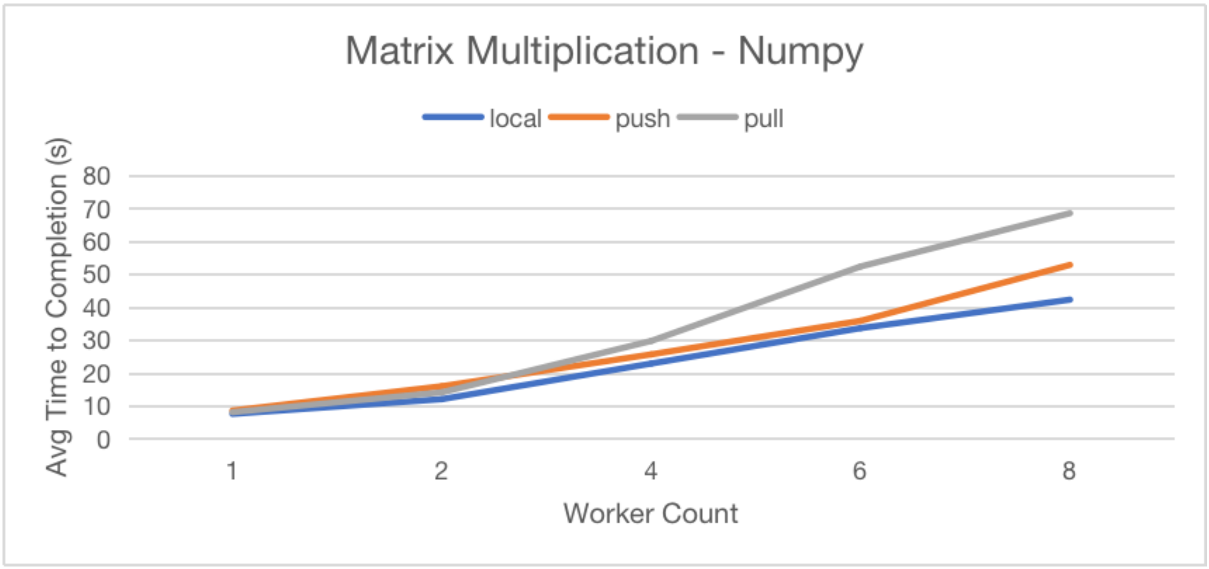
2. Indicates limitations in the parallelization strategy for the matrix multiplication task in Python.

Analysis:

1. Incorporating tasks with substantial computing costs inevitably raises the average time to completion. Simultaneously, the setup time for push workers tends to be overlooked, resulting in push workers demonstrating performance closely aligned with that of local workers. This observation underscores the commendable performance of our push workers, even when factoring in fault tolerance measures. This efficiency is particularly advantageous in the context of our load-balancing strategy and the seamless, non-blocking exchange of messages between push workers and task dispatchers. The effective coordination between these components further accentuates the overall robustness and performance of our system.
2. Nevertheless, our pull workers exhibit suboptimal performance under these circumstances, and this can be attributed to various factors. Primarily, the design of our pull workers relies on a polling strategy to receive tasks, introducing a timeout element that contributes to overall system costs. Furthermore, the completion of a task requires two round-trip communications, adding to the processing overhead. Additionally, the repetitive querying of the task dispatcher by the pull worker contributes to an increased operational cost.

Table 3 Floating Point Math (Numpy):

workers	local	push	pull
1	7.5188	8.5285	8.0435
2	12.0009	15.9332	14.0766
4	22.8281	25.6158	29.7018
6	33.5916	35.8146	52.337
8	42.2478	52.8103	68.5205



Floating Point Math (Numpy):

All Approaches:

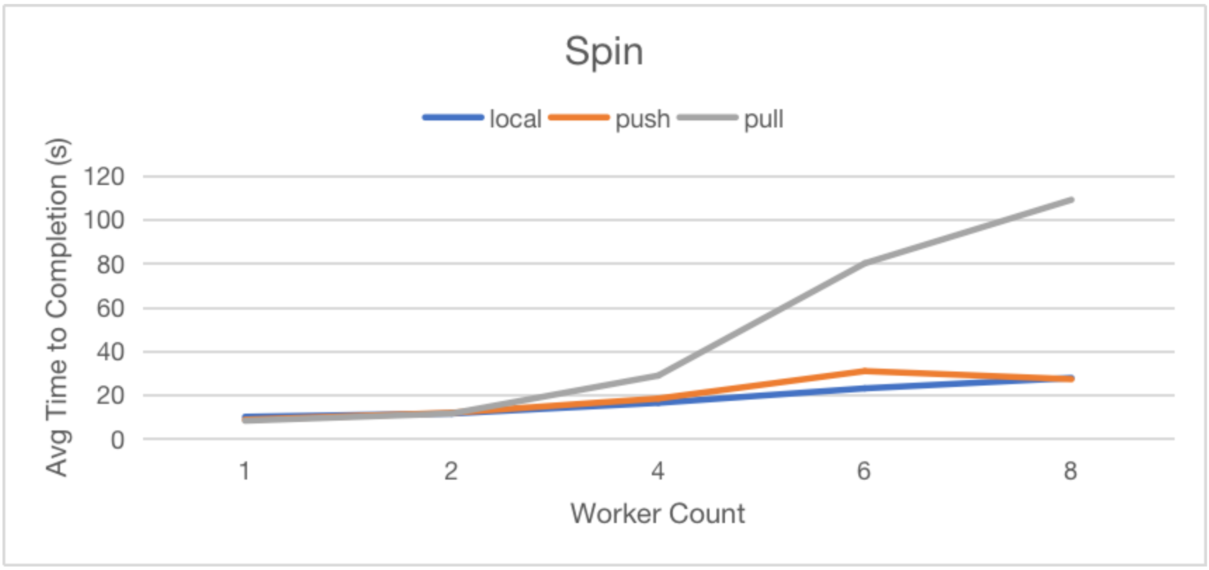
1. Poor scalability similar to the Python approach, with an increase in average time per task as the number of workers increases.
2. Numpy's design, limiting parallelism locally, contributes to the observed poor scaling.

Analysis:

1. Integrating tasks with significant computational demands naturally extends the average completion time. Concurrently, the setup duration for push workers often goes unnoticed, leading to push workers exhibiting performance comparable to that of local workers. This highlights the noteworthy efficiency of our push workers, even when considering fault tolerance measures. This effectiveness proves especially beneficial within the framework of our load-balancing approach and the smooth, asynchronous communication of messages between push workers and task dispatchers. The adept coordination between these elements further enhances the overall resilience and performance of our system.
2. Utilizing the matrix multiplication function from the NumPy library is the chosen approach. However, it's important to note that NumPy is optimized to leverage the full processing power of the CPU for diverse operations. Consequently, the scalability of this operation is notably limited during local testing. This limitation arises because NumPy tends to monopolize the CPU, impeding the concurrent execution of numerous other processes even on systems with multiple cores. Therefore, the cost of each task will be greater than the cost when only using Python matrix multiplication.

Table 4 Non-floating Point Math:

workers	local	push	pull
1	10.0247	8.7316	8.247
2	11.5174	11.8733	11.523
4	16.3839	18.3462	28.8308
6	23.0026	30.8746	80.075
8	27.7415	27.1947	109.0502



Non-floating Point Math:

Local and Push Approaches:

1. Show improvement in scalability, with a decrease in average time per task as the number of workers increases.
2. Indicates the effectiveness of these approaches in handling non-floating point math tasks.

Pull Approach:

1. Exhibits a substantial increase in average time per task with more workers.
2. Potential inefficiencies or increased communication overhead in the pull approach for non-floating point math tasks.

Analysis

1. Incorporating tasks that require substantial computational resources inherently prolongs the average time for task completion. Meanwhile, the setup time for push workers, often overlooked, contributes to push workers demonstrating performance akin to that of local workers. This underscores the significant efficiency of our push workers, even when factoring in fault tolerance measures. This efficacy is particularly advantageous within the context of our load-balancing strategy and the seamless, asynchronous communication of messages between push workers and task dispatchers. The proficient coordination among these components further elevates the overall resilience and efficiency of our system. In addition, our push workers even outperform the local workers when they have more workers.
2. However, our pull workers demonstrate less-than-optimal performance in such situations, and this can be ascribed to several factors. Firstly, the architecture of our pull workers is dependent on a polling strategy for task reception, introducing a timeout element that contributes to the overall costs of the system. Moreover, the task completion process necessitates two round-trip communications, increasing the processing overhead. Additionally, the frequent querying of the task dispatcher by the pull worker adds to the operational costs.

Conclusion:

1. The weak scaling study highlights the performance characteristics of our distributed FasS system across various tasks and worker configurations.
2. While some tasks demonstrate positive scaling trends, others show limitations in scalability.
3. Further investigation into the inefficiencies observed in specific tasks, especially in the Floating Point Math operations, is recommended to optimize performance.
4. In the future, we will set up tests when some nodes are down for performance on fault tolerance.