Modin OpenMPI Compute Engine

Andrew Zhang
Richard Lin
Sean Meng
Crystal Jin

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-265
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-265.html

December 17, 2021

Acknowledgement

I would like to thank Richard Lin, Crystal Jin, and Sean Meng for helping with the initial
research of the paper. I would also like to thank Devin Petersohn for providing valuable insight
and guidance for this project. Special thanks to Divij Sharma for proofreading assistance.

Modin OpenMPI Compute Engine

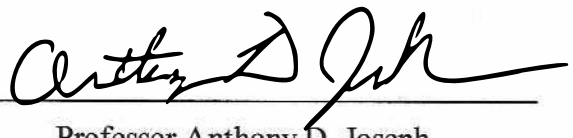
by Andrew M. Zhang

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

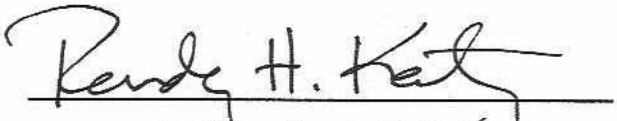


Professor Anthony D. Joseph
Research Advisor

17 December 2021

(Date)

* * * * * * *



Professor Randy H. Katz
Second Reader

17 December 2021

(Date)

Abstract

Modin OpenMPI Compute Engine

By

Andrew Zhang, Richard Lin, Crystal Jin, Sean Meng

University of California, Berkeley

Professor Anthony Joseph

Pandas is a popular dataframe manipulation tool used by data scientists. A key problem with Pandas is its inability to scale across cores, which severely limits its ability to deal with big data workloads. In order to keep up with ever larger datasets, data scientists need a dataframe tool that can scale effectively but also retain Pandas's ease of use. Modin, a drop-in substitute for Pandas, can effectively parallelize dataframe workloads and supports various computational backends, such as Ray, Dask, or Python. In this project we implement another compute backend for Modin: OpenMPI, an implementation of Message Passing Interface. This will allow users to tap into OpenMPI infrastructure to scale up their dataframe processing needs.

1 Introduction

Pandas is a popular Python library amongst data scientists for manipulating dataframes. Pandas operates relatively performantly on single machines by implementing dataframe operations as calls to other well-parallelized libraries (such as Numpy) under the hood. However, Pandas itself does not implement parallelism in its calls to underlying libraries, nor does it offer a way to use parallel computing solutions, such as multiple simultaneous processes, to parallelize work across a single machine or cluster. Thus Pandas cannot be scaled effectively.

Modin[5], a drop-in substitute for Pandas, aims to solve this problem. Modin splits a dataframe into multiple partitions that may be distributed across multiple systems (or in the case of a single machine, multiple cores), which allow for drastically increased performance that scales with computing resources. Another advantage of Modin is its modularity. The query compiler is implemented as a separate interface from its underlying execution engine. This means that Modin operations can be sent to various compute fabrics under the hood, including Ray[4], Dask[8], or plain Python (see Fig. 1).

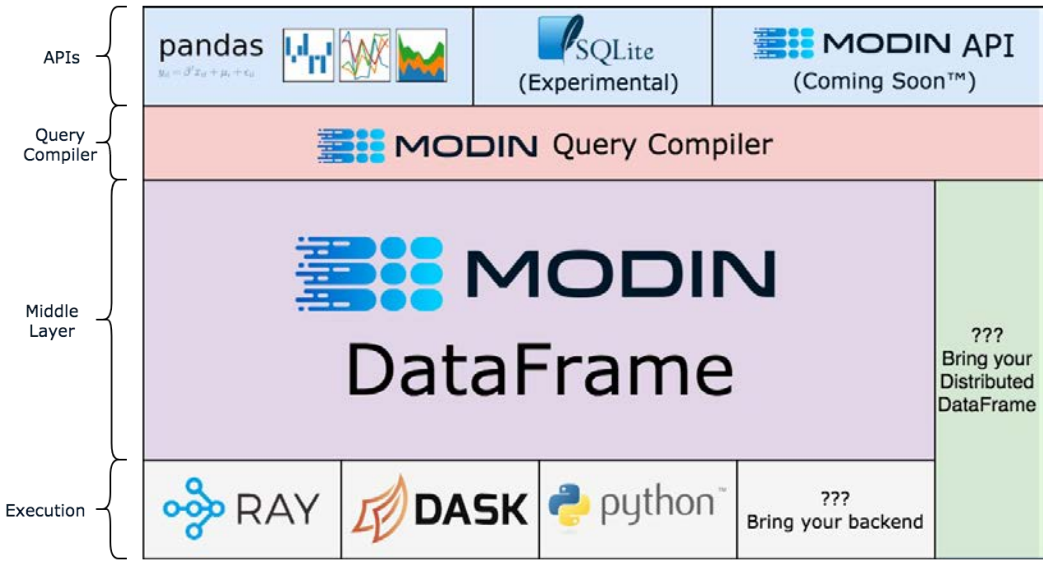


Figure 1: Architecture of Modin. Modin separates its implementation into several abstraction layers: APIs, Query Compiler, Middle Layer, and Execution. The goal is to provide access between layers so compute engines can be easily substituted

The current set of supported backends is somewhat limited. In this project, we implement a new execution engine using OpenMPI[2], a particular implementation of the Message Passing Interface(MPI)[3]. This will allow users who already have access to an OpenMPI optimized

environment to take advantage of Modin without having to change their computational infrastructure.

To summarize, our contribution is as follows:

- Implement an OpenMPI compute backend for Modin that takes full advantage of the features offered by the Message Passing Interface.

2 Background

2.1 Scaling in Python

Python has established itself as the data scientists' preferred language. There are numerous advantages to using Python for data science tasks, such as having a healthy suite of data manipulation tools and libraries including Jupyter Notebook, Numpy, and Pandas. Python is also more readable compared to other popular languages.

For larger workloads however, Python and popular data libraries run into problems. Pandas and Numpy do not have native support for executing across a compute cluster, which leaves the data scientist to either scale up their individual machine, which can only be done to a limit, to scale down their data, or to accept long execution times. Although Numpy does have underlying support for multithreading via its C extensions, computational speedups from native Python multithreading are hindered by the global interpreter lock (GIL), which prevents two lines of Python bytecode from being executed simultaneously. Thus multithreading speedups are limited to operations that can release the GIL, such as disk/network IO. Most developers opt to start new Python processes (each process will have its own GIL) in order to achieve parallel compute speedups. Despite the challenges, numerous Python libraries have emerged to help data scientists add scalable compute to their Python workloads.

Modin currently supports using Dask, Ray, or Python for its underlying compute engine. Although Ray and Dask are both designed for Python distributed computing, they differ in implementation and feature set which can lead to one framework being preferred over the other on the basis of performance, application constraints, or ease of use.

2.2 Dask

Dask is an open source low-level scheduler that can parallelize Python code as individual processes across clusters of machines or on a single machine. Dask.Distributed, a centrally managed, distributed, dynamic task scheduler, is also Modin's default choice for its scheduler. Task execution is coordinated by the scheduler and performed by worker processes. The

scheduler is also asynchronous and event-driven via futures that implement Python's concurrent.futures interface. Dask is fault tolerant, handling failure and addition of workers gracefully. According to the Dask documentation, "Dask is routinely run on thousand-machine clusters to process hundreds of terabytes of data efficiently within secure environments"[8].

2.3 Ray

Ray is an open-source library that provides an API for distributing applications across one machine or clusters of machines, similar to Dask. Modin uses Ray's Remote API. Remote API allows functions to run asynchronously as Tasks and class instances to run as remote processes called Actors. Unlike naive processes-based parallelism with native Python, Ray's Actor abstraction allows sharing mutable state across Tasks, which avoids duplication of data and expensive initialization overhead. Ray implements a distributed bottom-up scheduling scheme in which tasks are submitted to local schedulers, and a global scheduler is used to load balance across machines in the cluster.

Centralized scheduling versus bottom-up scheduling is one of the main differences between Dask and Ray. Ray's bottom-up scheduling allows for lower task latency and higher throughput as not all jobs must pass through a single centralized choke.

2.4 MPI Motivation

MPI is a standard for message passing that is common in the high performance computing industry. MPI itself does not implement parallel processing, rather it is achieved via multiple processes on one or more machines performing computation and coordinating between themselves via the communication abstraction that MPI provides. Unlike Dask or Ray, MPI does not provide a scheduler, meaning that complex parallelism outside of basic scatters and gathers requires additional user implementation on top of MPI calls[3]. The more primitive, bare bones nature of MPI allows for extremely low message passing overhead. Particular implementations of MPI can also take advantage of features like remote direct memory access (RDMA) to further increase performance.

We believe that by implementing an MPI Engine, we can benefit data scientists with access to existing MPI-based compute environments. For this project we use the OpenMPI implementation.

3. MPI Engine Overview

3.1 Design

Modin separates its internal implementation into several abstraction layers: APIs, Query Compiler, Middle Layer, and Execution. Modin aims to provide a clean separation between layers to allow for easy substitution for any particular layer. This is particularly important for the "bring your own backend" (see Fig. 1). This clean separation will allow us to focus on implementing the compute engine and provides us with the exact API the upper frameworks will call to interact with our OpenMPI Engine.

More specifically, our new OpenMPI Engine must implement the following base classes enumerated in Table 1.

Base Class	**Description**
IO	Reads in different file formats: csv, json, parquet, etc.
Axis Partition	Holds the data and metadata for data along an axis i.e. row or column. The API of this class is used by the frame manager to compute on the data in the partition.
Partition	Holds the data and metadata for a block of data in the dataframe. The API of this class is used by the frame manager to compute on the data in the partition.
Frame (DataFrame)	Holds the partitions that make up the dataframe and contains the frame manager instance. Applies functions across the partitions using the frame manager.
Frame Manager	Contains the logic for distributing computation across partitions. It is called by the DataFrame for applying functions.

Table 1: Base classes in Modin that were implemented to create a new execution engine and a description of what they do.

Once these classes are implemented, the MPI Engine can be initialized in Modin without modifications to any other layer or application code.

3.2 Implementation Details

To interact between OpenMPI and Python, we opted to use mpi4py[1], a Python wrapper library that is compatible with various MPI implementations. Our design of the MPI execution engine is

similar to the Dask execution engine, that is the MPI Engine will execute tasks asynchronously using a pool of worker processes across a single or multiple machines. More specifically, a single operation on a dataframe will be split into multiple tasks that each operate on a partition of the aforementioned dataframe in parallel. Then the MPI Engine will receive and submit tasks for asynchronous execution to a main executor process through mpi4py.futures. The mpi4py.futures package is based on Python's native concurrent.futures package. The main executor will spawn off worker processes using dynamic process management, a feature specified by the MPI-2 standard. Dynamic process management allows the main executor to create additional processes during runtime after startup as opposed to the number of processes being fixed at startup.

3.3 MPI Pool Executor Lifecycle

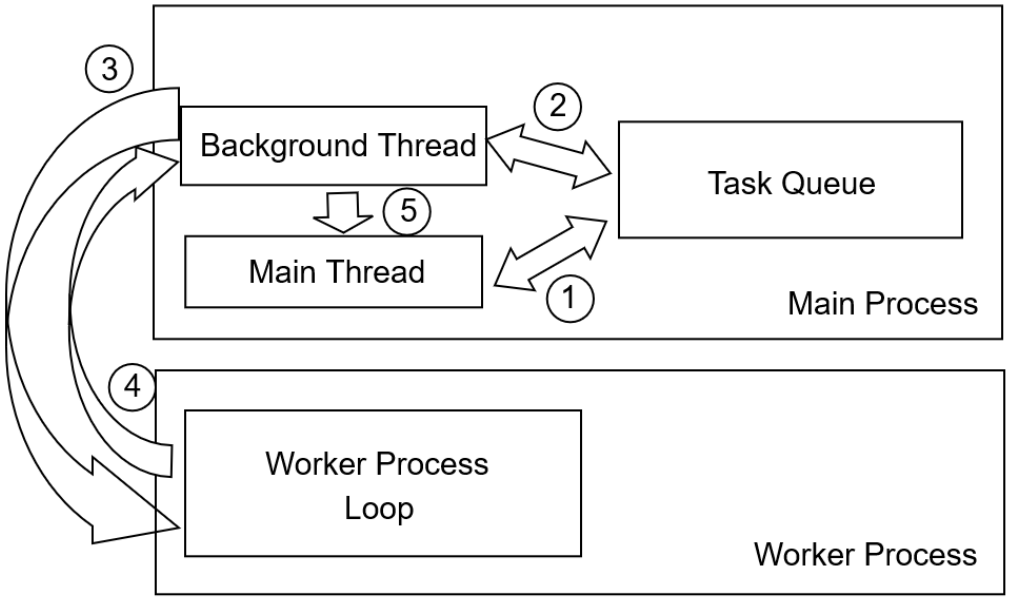


Figure 2: This diagram illustrates how tasks, which consist of a function and its arguments, are submitted to the MPI Process Pool for asynchronous execution. We start with the worker process already being initialized by the Background Thread. 1) The main thread places a task onto the task queue and receives a future. 2) Background thread dequeues the task 3) Background Thread serializes the task and MPI Sends it to a Worker Process. 4) The Worker Process receives the task, deserializes it, executes the function on the arguments, serializes the results and calls MPI Send to send it back to the Background Thread 5) Background Thread receives the result, deserializes it, and calls the future's set_result method. The main thread can now retrieve this future's results

3.3.1 Background Thread Send

Upon initialization, the MPI Pool Executor creates a background thread whose sole purpose is to monitor a global task queue and submits tasks (a function and its arguments) to available workers. If the number of workers is below the max threshold, MPI will spawn an additional

worker via dynamic process management. Task submission involves serializing both the function and its arguments (in our case via cloudpickle) and sending it to an available worker via MPI Send. After the send, the thread marks the worker as busy so as to not send it any additional tasks.

3.3.2 Worker Process

Workers repeat the following loop for the duration of the program: 1) wait to receive a task from the executor background thread 2) upon receive, deserialize the task function and arguments 3) run the function on the arguments 4) serialize the results and send them back to the background thread via MPI Send.

3.3.3 Background Thread Receive

The background thread will occasionally call MPI Iprobe, to check if there are any available messages from the worker processes. If there is a message to be received, the background thread will deserialize the result, fetch the appropriate future object and call future.set_result. This method will mark the future as done, run all done callbacks, and place the execution result into the future object. This worker the result was received from will be marked as available by the background thread.

3.3.4 Additional Remarks

Note that the pool executor is a centralized task scheduler. So far we assumed that all future objects remain in the single main code flow execution process. Although this implements a functional MPI Engine, there are some inefficiencies with this naive design that we will address later.

3.4 Challenges

3.4.1 Asynchronous Function Execution on Future Object Arguments

```
0 # Function signatures
1 func1(arg: pd.DataFrame) -> pd.DataFrame
2 func2(arg: Future) -> pd.DataFrame
3 func3(arg: Future) -> pd.DataFrame
5 # Asynchronous executions
6 future_result_1 = mpi_executor.submit(func1, df)
7 future_result_2 = mpi_executor.submit(func2, future_result_1) # This line may error
8 future_result_1.block_until_done() # Wait for future_result_1 to complete execution
9 future_3 = mpi_executor.submit(func3, future_result_1) # This line will not error
```

Listing 1: The first line of code generates a future representing the result of function func1 on non-future argument df. The second line attempts to asynchronously execute func2 on the future object future_result_1, which can lead to unexpected behavior at runtime depending on the state of the future_result_1 when this line is executed. The third line will lead to unexpected behavior, but will block until future_result_1 completes execution

Since not all Pandas operations can be translated into a single map reduce, the query compiler may need to execute functions whose arguments are futures (the alternative is to block until the future yields a result). In Code Listing 1, we document the problem with pseudocode. On line 6, future_1 refers to the asynchronous result of executing func1 on df. Line 7 attempts to run func2, which takes a future as an argument, on future_result_1. This operation will not succeed if future_result_1 is not done by line 7's execution. This is because futures cannot be trivially serialized due to its internal mutex lock and the fact that set_result will not be called correctly on the deserialized future by the pool's background thread as it has no knowledge of the existence of the deserialized future.

Even if we remove the mutex lock and re-implement futures such that they can be serialized without loss of functionality (see Sec 3.4.2), this would still pose a runtime problem. If future_result_1 has not completed execution by the time Line 7 is executed, future_result_1 will be serialized and sent to a worker for execution. This worker that is trying to run func2 on future_result_1 would be forced to block until future_result_1 is done executing. In the meantime, this worker must spin-wait.

We provide a polling based solution to this problem as shown on Line 8-9. Before submitting func3 to the executor, we block until its future object argument, future_result_1, is finished executing. The main drawback to this solution is that we are forced to block on line 8 and cannot continue code execution. Our actual implementation will only perform a periodic non-blocking check to see if the future is done, and use callbacks to place dependent tasks onto the task queue.

3.4.2 Serializable Futures (SFuture)

A typical future object cannot be trivially serialized due to aforementioned reasons; an internal mutex lock and that pool executor background thread will not call set_result on a deserialized future. However, for the purposes of implementing an MPI Engine for Modin, we do not need futures that must both be serializable and retain full functionality. We document an implementation of a reduced functionality serializable future that leverages MPI features below. For the purposes of clarity, we will refer to these serializable futures as SFutures.

3.4.2.1 Serialization Deserialization API

```
class SFuture(Future):
  # MPI Dynamic RMA Window
  win : MPI.Window = MPI.Win.Create_dynamic(MPI.INFO_NULL, comm=MPI.COMM_WORLD)

  def __init__(self, future=None, [...]):

    # Unserialized attributes. These stay on the host process.
    self.future : [None, Future] = future # Underlying future instance that we are wrapping.
    self.addr_and_sz = array.array('Q', (0, 0)) # Q denotes long long type.
                                                # First number stores the MPI RMA Window Address.
                                                 # Second number stores serialized dataframe length

    # Instance Attributes that will be serialized/deserialized on pickle/unpickle. This can be
    # achieved by overriding __get_state__ and __set_state__.
    self.uuid : str
    self.originating_process_num : int
    self.addr_and_sz_ptr : int
    [...]
    if new_SFuture_is__being_instantiated:
      self.uuid = uuid.uuid4()
      self.originating_process_num = MPI.COMM_WORLD.Get_rank()
      self.addr_and_sz_ptr = MPI.Get_address(self.addr_and_sz)
      SFuture.win.Attach(self.addr_and_sz)  # Attach the addr_and_sz tuple to the RMA window
    else:
      self.uuid, self.originating_process_num, self.addr_and_sz_ptr = load_values_from_unpickle()

    # Result cache
    self.result_cache : [pd.DataFrame, Any] = None # Results are cached to avoid redundant fetching
                                                   # from remote processes and deserialization
```

Listing 2: A pseudocode view of how SFutures are initialized. Ellipsis indicates omitted code.

In Listing 2, we list the SFuture's API. Note the constructor's behavior; if we attempt to initialize an SFuture with the same future_id as an existing SFuture in the current process's scope, we must receive the same SFuture object. This design is based on Dask's Serializable Lock implementation[11] (if a Serializable Lock is deserialized on its process of origin, you receive the exact instance of the original Serializable Lock). This means you cannot have two SFuture instances in the same process with the same future_id, since they would be the same object. One way to think of this is that within a process, an SFuture with the same future_id behaves like a

singleton. To implement this feature we overrode the class's metatype's __new__ dunder method to check a global dictionary in the process memory and either fetch the existing SFuture or continue to instantiate and initialize a new SFuture instance.

In order to ensure this feature also works on deserializing an SFuture, we override its __get_state__ dunder method such that it only returns its future_id, process of origin, and memory start location (see Listing 2). If the SFuture is deserialized in a process that already has an instance with the same future_id in its memory, we will get an alias to the original future; deserializing involves a call to __new__ and a call to __set_state__, the former method will return the original SFuture and latter will never have an effect on the future's state, since future_id, machine of origin, and memory start location never change once initialized. If such an instance does not exist, we must have deserialized the SFuture in a process from which the SFuture did not originate. In that case we set the state so this "remote" SFuture has a copy of the future_id, process of origin, and memory start location. We only serialize these three variables, as these are the only pieces of information needed to perform a remote result retrieval. We document the result retrieval process in the upcoming section.

3.4.2.2 Result API

```
def SFuture.result(self, timeout=None):
  # Check local cache for result. Code is omitted
  [...]
  # self instances is a  remotely deserialized SFuture.
  # We must fetch the result from the originating process
  addr_and_sz_temp = array.array('Q', (0, 0))
  while time_elapsed < timeout:
    # Start One-sided RMA Access Epoch
    SFuture.win.Lock(self.originating_process_num, lock_type=MPI.LOCK_SHARED)
    # Attempt to fetch the address of the serialized result and its length
    SFuture.win.Get(addr_and_sz, target_rank=self.originating_process_num,
                    target=self.addr_and_sz_ptr)
    # Block until Get succeeds
    SFuture.win.Unlock(self.originating_process_num)

    if addr_and_sz[0] == 0:
      # Result not ready. Wait for 100ms. Then try again.
      time.sleep(0.1)
      continue

    # Result is ready
    # Create a buffer to load in result
    buffer = create_buffer(addr_and_sz[1])
    # Start One-sided RMA Access Epoch
    SFuture.win.Lock(self.originating_process_num, lock_type=MPI.LOCK_SHARED)
    # Load the remote serialized result into local buffer
    SFuture.win.Get(buffer, target_rank=self.originating_process_num, target=addr_and_sz[0])
    # Block until done
    SFuture.win.Unlock(self.originating_process_num)

    # Cache the result
```

```
    self.result_cache = pkl.load(buffer)
    # Return result
    return self.result_cache
```

Listing 3: SFuture's pseudo-code implementation of the result() method. Ellipsis indicates omitted code. Note how the API leverages MPI's One-Sided RMA, which obviates the need for synchronization between processes to coordinate Sends and Receives.

Futures are never serialized with their result, even if the result is available. Thus a deserialized future must retrieve its result from its process of origin. We do this via MPI's one sided communication API. Note that a deserialized SFuture has an address for the length and location tuple of its remote serialized result. To retrieve the result (see Listing 3), we first do a one-sided Remote Memory Access (RMA) operation to determine the length of the serialized result. If the length is zero, we sleep and check again later. If the length is non-zero, we know the result is ready and the address part of the tuple is now a valid remote memory address. We fetch the location of the pickled result and read the pickled result out of the remote process memory.

3.4.2.2 Set Result API

```
def SFuture.set_result(self, result):
  # Check if the result is already set. Code is omitted
  [...]
  # API only supports setting SFuture result if setter is originating process
  assert self.originating_process_num == MPI.COMM_WORLD.Get_rank()

  self.result_buffer = result_to_buffer(result) # Serialize the result
  # Start One-sided RMA Access Epoch. Lock must be exclusive to ensure Get processes are not
  # simultaneously executing.
  SFuture.win.Lock(self.originating_process_num, lock_type=MPI.LOCK_EXCLUSIVE)
  # Add buffer to Dynamic RMA Window. Set the address and size values.
  SFuture.win.Attach(self.result_buffer)
  self.addr_and_sz[0] = MPI.Get_address(self.result_buffer)
  self.addr_and_sz[1] = self.result_buffer.length()
  # End access epoch
  SFuture.win.Unlock()

  # Set the result of the underlying future. This is necessary to ensure callbacks are invoked
  return self.future.set_result(result)
```

Listing 4: The set_result method pseudo-code. Ellipsis indicates omitted code.

The set_result method is rather simple (see Listing 4). We allocate a buffer and serialize our result into the buffer. Then, we start an RMA Access Epoch and attach the buffer to the MPI RMA Window, set the addr_and_sz pointer to contain the buffer's address and length, and end the RMA Access Epoch. We also make sure to call set_result on the Python future object that SFuture wraps. This will ensure that callbacks attached to the underlying Python future are executed correctly

3.4.2.3 API Motivation

There are several reasons why we chose this particular API implementation. First, by avoiding serializing the result into the SFuture, we ensure the messages we send via MPI Send are shorter and that the data transfer heavy lifting is done via RMA operations, which can be significantly faster than Send/Recv on systems with hardware support for RMA[10]. One-sided RMA also obviates the need to synchronize MPI Send and Receive calls between the process that has the data and the one that wants the data. Second, this allows us to passively implement serialization caching. Serializing data frames takes a non-trivial amount of time, and our implementation will avoid having to deal with repeat serializations.

4 Evaluation

4.1 Testing Scenario

In order to gauge the performance of our MPI Engine, we tested the speed of five Pandas operations: read_csv, count, isnull, apply, and groupby. For the data, we used the Dask NYC taxi dataset[9], which initially contains 12 million rows and is roughly 2GBs on disk. We ran strong and weak scaling benchmarks on an AWS Linux EC2 instance with 2, 4, and 8 and 16 logical cores. All tests allocate one process per core, except the 32 process test, which oversubscribes and allocates 2 MPI processes per logical core. Note that MPI will reserve 1 of its processes for its main thread/pool manager (ie 2 processes total means 1 worker process and 1 process that steps through the application code). We ran each test 5 times and took the average execution time. For the strong scaling benchmarks, we kept the number of rows the same for each test. For weak scaling, we scaled up the data with the cores proportionally. We also ran the same benchmarks on Pandas as a baseline with the exception that we did not change the number of cores, because Pandas does not support multi-core parallelism.

4.2 Strong Scaling

For the strong scaling benchmark, we scale the number of cores while we hold the CSV size steady at 2GB. In the plots below we also provide a native Pandas bar as a performance baseline.

4.2.1 Read CSV

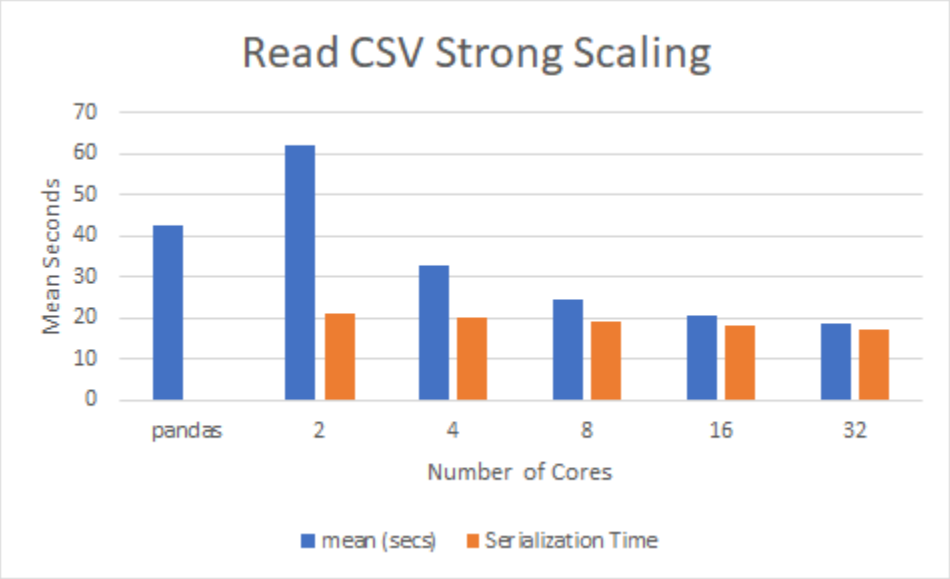


Figure 3: Read CSV strong scaling benchmarks. The first bar is the Pandas baseline. For all other bars, core count doubles per x-tick. The y-axis plots the mean number of seconds the test took to execute across 5 separate runs. The orange bars show time spent serializing.

As we can see in Fig. 3, MPI does exhibit the expected scaling pattern as we double the number of cores while holding dataframe size steady. That is, we see that execution time roughly halves with each doubling of cores, at least initially. We see that by utilizing all available cores on the node in MPI, we can achieve a roughly 2x speedup vs native Pandas (42s to 19s). Because read csv passes around the full dataframe, this operation incurs heavy costs to serialization and deserialization overheads. From the orange bars in Fig. 3, we see that the serialization time, which fails to strongly scale with core count, bottlenecks MPI Engine's ability to scale the overall operation.

The non-scaling serialization costs have to do with the MPI Engine's centralized pool executor setup. The pool executor must receive all serialized results from its worker processes and deserialize them one at a time. For instance, for the 8 process benchmark, the main process must deserialize one seventh of the full dataframe 7 times (the 8 process benchmark has 7 workers). We omit the serialization time bar in other graphs, as they do not spend a majority of their time in serialization.

4.2.2 Count

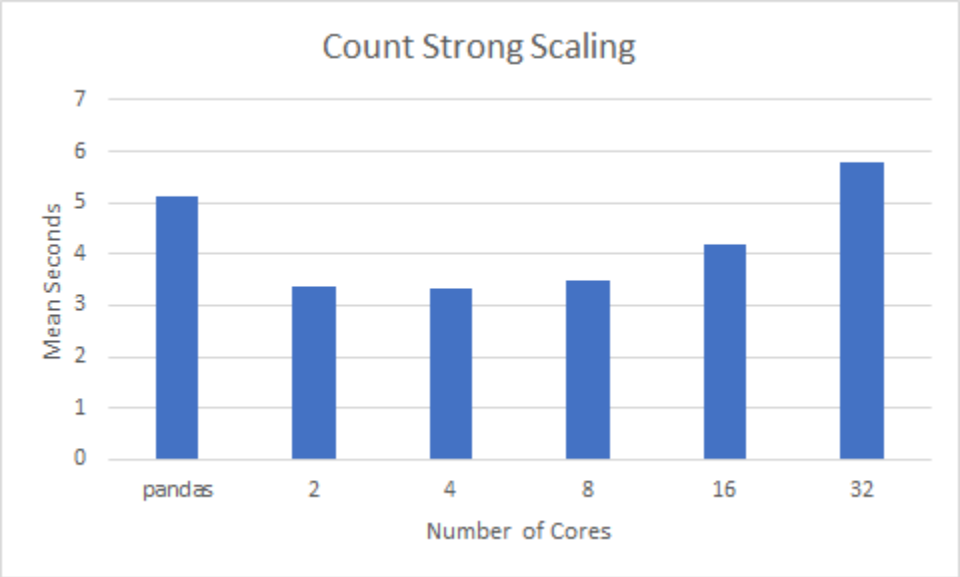


Figure 4: Count strong scaling benchmarks. The first bar is the Pandas baseline. For all other bars, core count doubles per x-tick. The y-axis plots the mean number of seconds the test took to execute across 5 separate runs.

The count benchmark (see Fig. 4), shows some unusual behavior. Namely the scaling outperforms native Pandas, but additional cores appear to gradually increase the runtime. We believe this may be related to serialization times. If we compare Pandas and the best MPI core count runtime, 1.5x speedup (5.2s to 3.4s).

4.2.3 IsNull

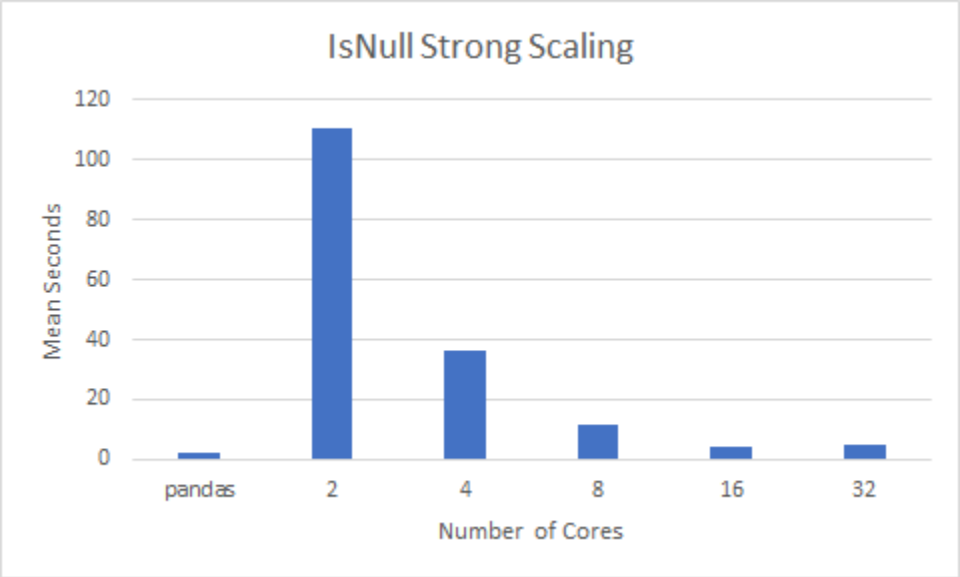


Figure 5: IsNull strong scaling benchmarks. The first bar is the Pandas baseline. For all other bars, core count doubles per x-tick. The y-axis plots the mean number of seconds the test took to execute across 5 separate runs.

IsNull (see Fig. 5) also exhibits effective strong scaling, however due to serialization overhead, it cannot outperform native Pandas and in fact achieves a roughly 2.2x slowdown (1.8s to 4.2s).

4.2.4 Apply

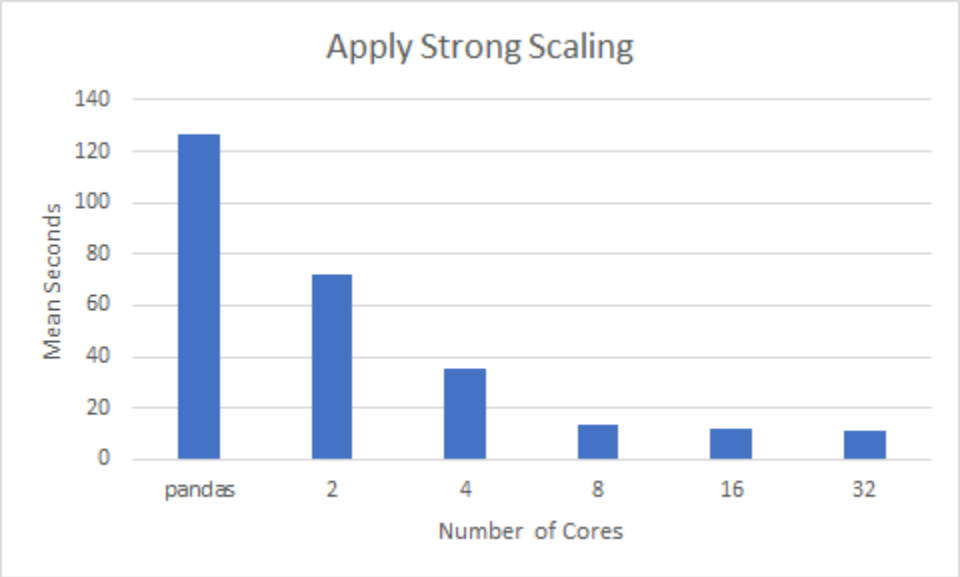


Figure 6: Apply strong scaling benchmarks. The first bar is the Pandas baseline. For all other bars, core count doubles per x-tick. The y-axis plots the mean number of seconds the test took to execute across 5 separate runs.

We see the most significant improvements versus the Pandas baseline in the apply operation (see Fig. 6). This is expected as apply is implemented as a for loop under the hood, and unlike read csv, does not take advantage of a C-based parsing engine. This makes the apply operation very inefficient, and parallelization yields significant gains. The parallel scaling only carries up to 8 cores before tapering out. Nonetheless, we achieve over 12x speedup relative to native Pandas (127s to 10s).

4.2.5 Groupby

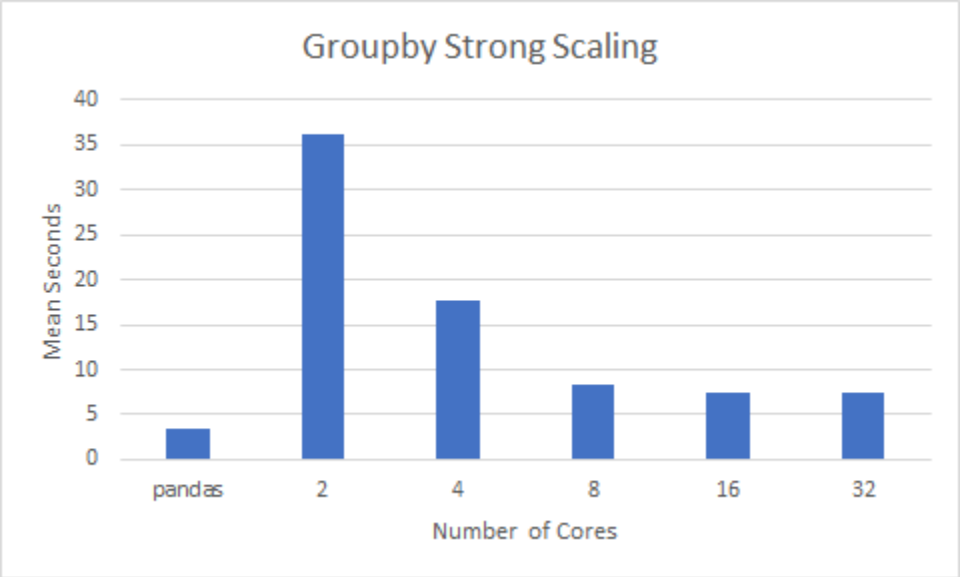


Figure 7: Groupby strong scaling benchmarks. The first bar is the Pandas baseline. For all other bars, core count doubles per x-tick. The y-axis plots the mean number of seconds the test took to execute across 5 separate runs.

Although groupby does exhibit scaling efficiency initially, speedups taper off after 8 cores and do not outperform native Pandas (see Fig. 7). Unfortunately between Pandas and the fastest MPI core count, we see a roughly 2.1x slowdown (3.5s to 7.4s).

4.3 Weak Scaling

For the weak scaling benchmarks, we double the size of the CSV as well as the number of processes every iteration. We also provide a Pandas performance baseline. Since Pandas does not support changing process count, we only scale the CSV size and we do not expect Pandas to weak scale at all. This expectation is confirmed in the plots in Sec. 4.3. Note that the graphs below are plotted on the log seconds scale on the y-axis. Ideal weak scaling on a log seconds plot will appear as bars maintaining their height; this indicates that as core counts and data size double, execution time remains roughly the same. Since the original dataset was only 2GB, for the 32 process/4GB test, we resampled the original dataset to double its size to 4GB.

4.3.1 Read CSV

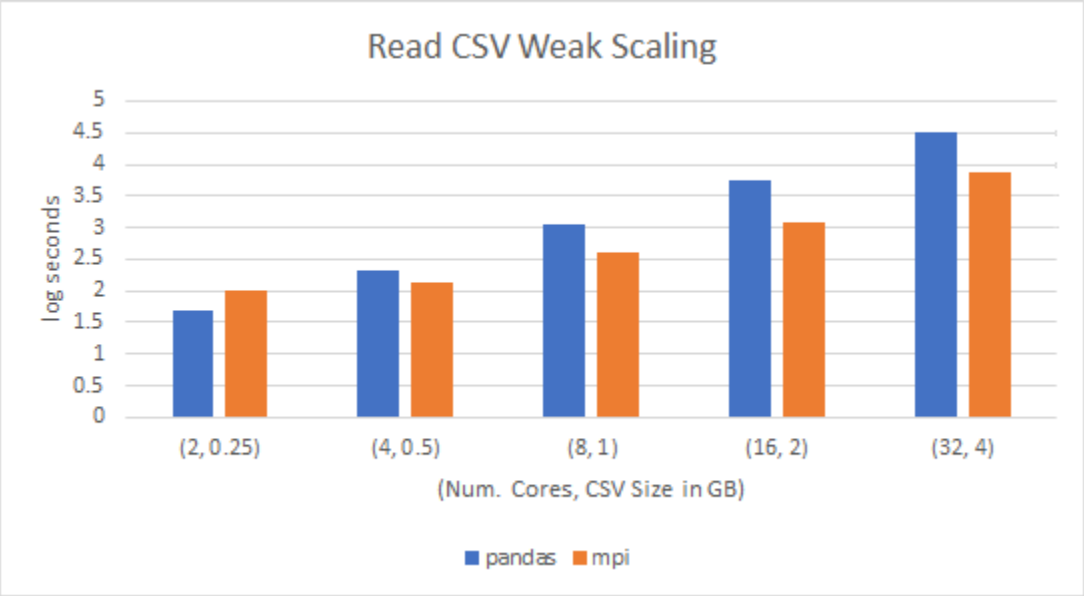


Figure 8: Read CSV weak scaling benchmarks. The x-ticks denote (MPI process count - if applicable, CSV size in GB). The blue bars represent log second Pandas execution times. The orange bars represent log second MPI execution times.

The read csv operation does not weak scale well, although it does outperform native Pandas as it scales up (see Fig. 8). The benchmark exhibits some weak scaling for 2 through 8 cores, but fails to continue weak scaling after.

4.2.2 Count

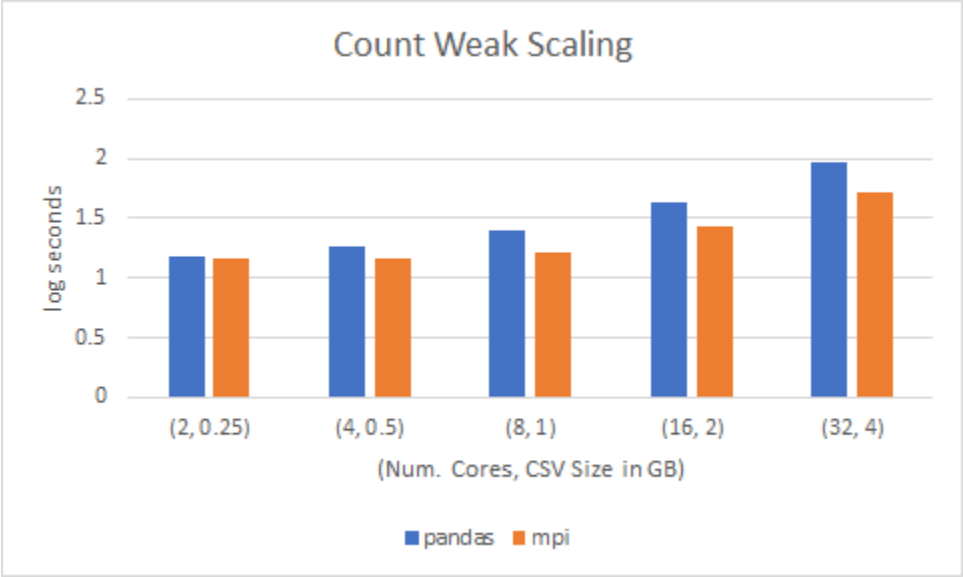


Figure 9: Count weak scaling benchmarks The x-ticks denote (MPI process count - if applicable, CSV size in GB). The blue bars represent log second Pandas execution times. The orange bars represent log second MPI execution times.

Count fails to weak scale perfectly, however it can outperform native Pandas in all benchmarked cases (see Fig. 9).

4.2.3 IsNull

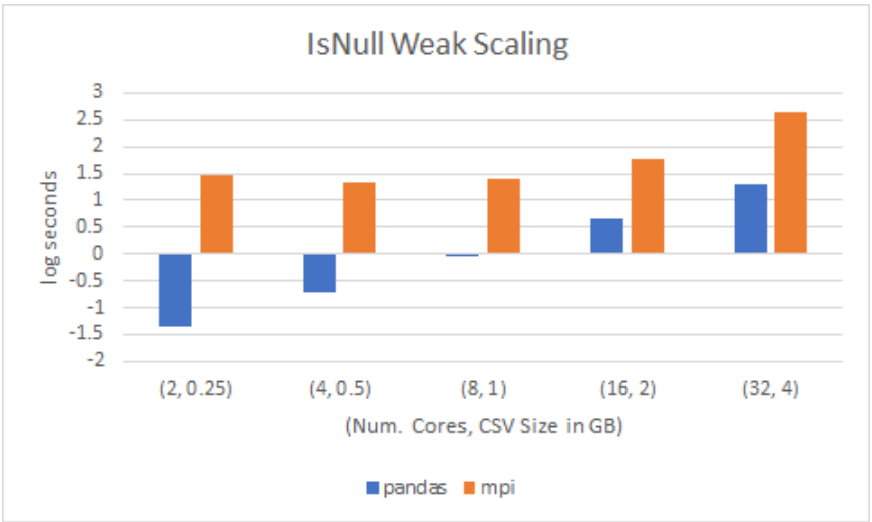


Figure 10: IsNull weak scaling benchmarks. The x-ticks denote (MPI process count - if applicable, CSV size in GB). The blue bars represent log second Pandas execution times. The orange bars represent log second MPI execution times.

IsNull demonstrates reasonable weak scaling by having similar execution times between 2 and 16 processes (see Fig. 10). However, it fails to continue weak scaling on the oversubscribed 32 process case. As with all other oversubscribed benchmark cases, assigning more processes than there are cores never has the effect of significant performance increase.

4.2.4 Apply

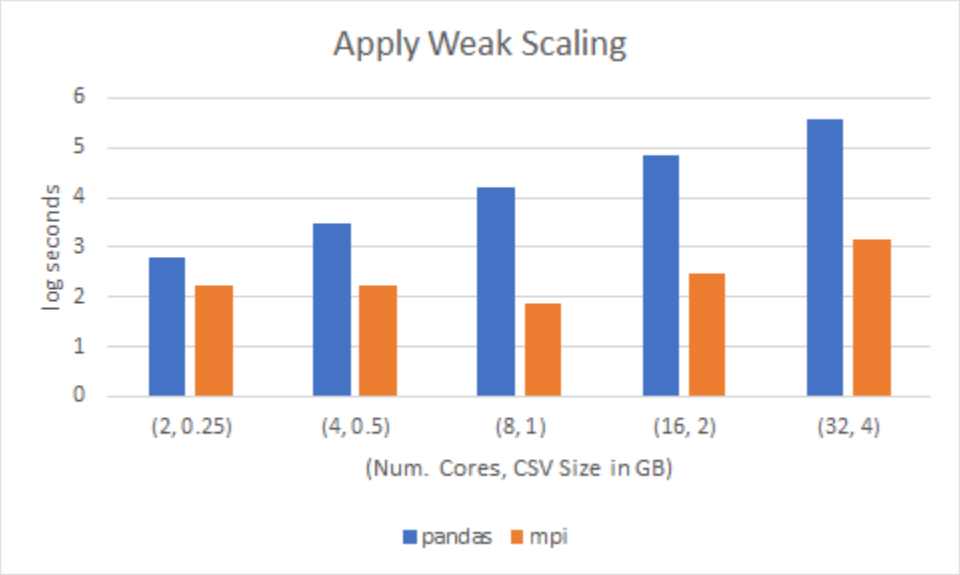


Figure 11: Apply weak scaling benchmarks. The x-ticks denote (MPI process count - if applicable, CSV size in GB). The blue bars represent log second Pandas execution times. The orange bars represent log second MPI execution times.

Apply demonstrates reliable weak scaling up to 16 processes and significantly outperforms native Pandas speeds (see Fig. 11). The oversubscribed 32 process case weak scales slightly poorly compared to the other apply benchmarks.

4.2.5 Groupby

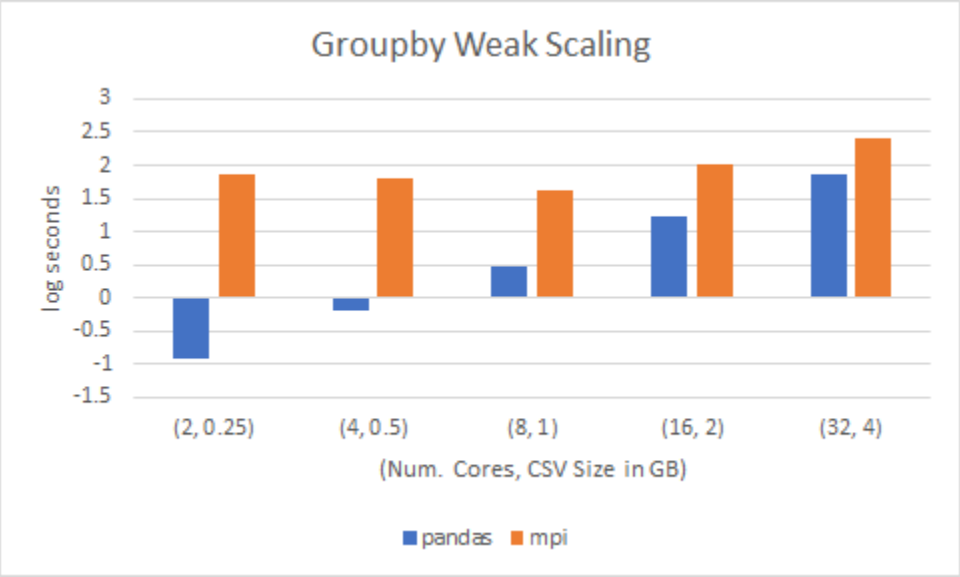


Figure 12: Groupby weak scaling benchmarks. The x-ticks denote (MPI process count - if applicable, CSV size in GB). The blue bars represent log second Pandas execution times. The orange bars represent log second MPI execution times.

Groupby, does exhibit consistent weak scaling up to 16 processes. Like previous benchmarks, the oversubscribed case weak scales more poorly (see Fig. 12). Despite consistent weak scaling, groupby is unable to outperform native Pandas.

5 Conclusion

Overall the MPI Engine demonstrates a moderate ability to weak and strong scale. From the plots above, we see that the MPI Engine performed the best on the apply case, outperforming native Pandas significantly and demonstrating consistent strong and weak scaling. Since apply is implemented as a single threaded for loop in Pandas, the MPI Engine could take full advantage of its parallelism without losing ground to overhead costs. The MPI Engine demonstrated consistent strong scaling in nearly all cases except count. Though for isnull and groupby, the MPI Engine was unable to outperform native Pandas benchmarks. We believe this is due to the parallelism advantage to overhead ratio. Pandas, unlike our MPI Engine, does not need to split and serialize a dataframe in order to begin operations. Furthermore, other researchers have noted that MPI does have its own overhead costs[7], aside from the ones our MPI Engine introduces. If the Pandas operation is more or less just passed to Numpy, which is parallelized in its C extension, Pandas will be highly performant as it's essentially running fully parallelized and does not need to deal with additional serialization costs. Meanwhile if the Pandas operation does not pass the data to some underlying parallelized C extension, then MPI's parallelized speedups will

be larger than its overhead costs, allowing our engine to outperform Pandas. This theory is consistent with the fact that read csv and apply exhibit large speedups, count is significantly slowed down by serialization costs, and all other benchmarks strongly scale relatively well despite not outperforming Pandas.

6 Future Work

The MPI Engine implemented in this project is relatively simple. Our implementation uses a central executor process that could be a bottleneck on larger workloads, both network wise and computationally. Future work that branches off of this initial implementation might want to explore more complicated and commonly-used MPI compute fabrics and features.

One of the main costs that hinder performance benefits is the serializer. In this project we opted to use cloudpickle as our object serializer. Passing dataframes and execution tasks between processes involves serializing the function and the dataframe. Although we were able to save time by caching results to avoid going through the serialization/deserialization process unnecessarily, there is likely a better way to serialize dataframes and pass them between machines. This improvement may be related to exploring more complicated MPI compute fabric configurations that take into account data locality and avoid unnecessary dataframe passing. The only reason we need to serialize the data is because Pandas dataframes are not contiguous in memory and MPI requires data that is to be transferred to be contiguous. Thus, another way to improve serialization performance may involve tapping the underlying Numpy arrays inside the Pandas dataframe, as Numpy arrays are contiguous in memory.

Finally, we see that not all Pandas operations benefit from parallelization in MPI. Aside from reducing overhead costs of the engine, future work may be done to have Modin or the MPI Engine short circuit and compute in native Pandas a trivial operation is being performed. This should allow Modin and the MPI Engine to achieve the best of both worlds; if the operation is not bottlenecked by slow Python execution, parallelize with Modin and MPI otherwise execute in native Pandas. This feature would allow our framework to leverage parallelism if and only if the speedups outweigh the overhead costs.

7 Acknowledgements

I would like to thank Richard Lin, Crystal Jin, and Sean Meng for helping with the initial research of the paper. I would also like to thank Devin Petersohn for providing valuable insight and guidance for this project. Special thanks to Divij Sharma for proofreading assistance.

Works Cited

[1] Dalcin, Lisandro. “MPI for Python.” *MPI for Python - MPI for Python 3.0.3 Documentation*, 2020, mpi4py.readthedocs.io/en/stable/index.html.

[2] Gabriel, Edgar, et al. "Open MPI: Goals, concept, and design of a next generation MPI implementation." European Parallel Virtual Machine/Message Passing Interface Users’ Group Meeting. Springer, Berlin, Heidelberg, 2004.

[3] Gropp, William, et al. Using MPI: portable parallel programming with the message-passing interface. Vol. 1. MIT press, 1999.

[4] Moritz, Philipp, et al. "Ray: A distributed framework for emerging {AI} applications." 13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18). 2018.

[5] Petersohn, Devin, and Anthony D. Joseph. Scaling Interactive Data Science Transparently with Modin. Tech. rep. Electrical Engineering and Computer Sciences, University of California at Berkeley, 2018.

[6] Petersohn, Devin. “Scale Your Pandas Workflow by Changing a Single Line of Code.” *Scale Your Pandas Workflow by Changing a Single Line of Code - Modin 0.8.2 Documentation*, 2020, modin.readthedocs.io/en/latest/index.html.

[7] Raffenetti, Ken, et al. "Why is MPI so slow? analyzing the fundamental limits in implementing MPI-3.1." Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. 2017.

[8] Rocklin, Matthew. "Dask: Parallel computation with blocked algorithms and task scheduling." Proceedings of the 14th python in science conference. No. 130-136. 2015.

[9] Rocklin, Matthew. *Distributed Pandas on a Cluster with Dask DataFrames*, matthewrocklin.com/blog/work/2017/01/12/dask-dataframes.

[10] Gropp, William. “Lecture 34: One-sided Communication in MPI”. CS598-s15 Designing and Building Applications for Extreme Scale Systems. University of Illinois, Urbana-Champaign. http://wgropp.cs.illinois.edu/courses/cs598-s15/. Accessed 9 Dec. 2021.

[11] Dask Core Developers https://docs.dask.org/en/latest/_modules/dask/utils.html Accessed 9 Dec. 2021.