Issue #1 - Alina, Gabriel, Mir, Howard

DOCUMENT

This document describes the implementation details for fixing issue https://github.com/pandas-dev/pandas/issues/40956 in pandas library.

Team Members

The following team members were involved in fixing the issue: Alina, Gabriel, Mir, Howard

Implementation- Howard, Alina

Testing - Gabriel, Mir

Documentation - Alina, Gabriel, Mir, Howard

Description

- The issue revolves around slow performance with the pandas DataFrame.corr() method when using the "spearman" or "pearson" correlation method.
- They suggest adding an optional argument to parallelize the computation.
- The underlying implementation of these different methods are done in an extension of the python language called Cython.

Implementation

- The new multi-threaded implementation must be done in the Cython `*.pyx` files since the original single-thread implementation is written in Cython for performance reasons.
- A built-in Cython library called `cython.parallel` is used for multi-threading
- In the original implementation, the correlation matrix calculation is done inside methods `nan_corr` and `nancorr_spearman` in `pandas/pandas/_libs/algos.pyx`. They each have double nested for-loops where for each method, a symmetric matrix `result` is filled in cell by cell to compute the resulting correlation matrix. These are prime candidates for parallelization as each iteration does not depend on any of the previous iteration.
- Separate functions were created to solely handle the parallel aspect of each implementation called `pnan_corr` and `pnancorr_spearman`.
- `pnan_corr` uses a statistical algorithm called “Welford's online algorithm” this was abstracted out to an external function to prevent duplicate code.
- An additional optional Boolean parameter called “parallelize” is added to the `DataFrame.corr()` function signature that indicates whether or not the multithreading implementation is used. From there the respective implementation is called (ie. `nan_corr` or `pnan_corr`, `nancorr_spearman` or `pnancorr_spearman`).

Changes Made

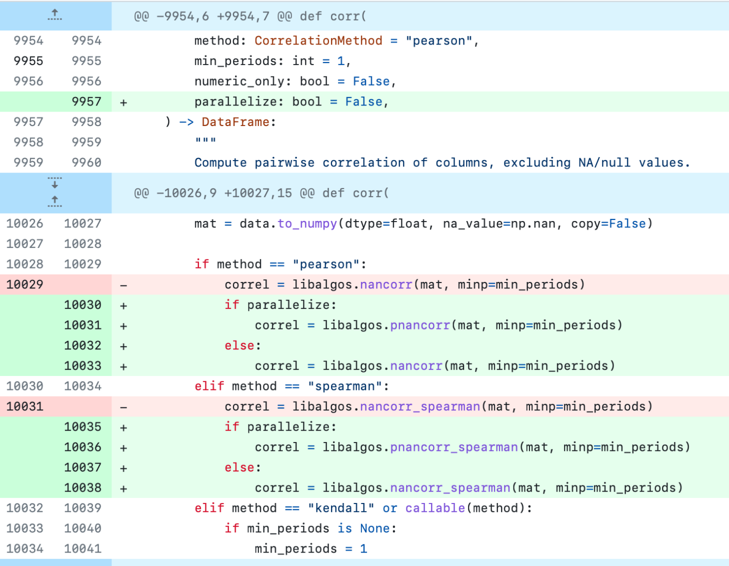
The following files were modified as a result of the implementation:

- `pandas/pandas/core/frame.py`

- `pandas/pandas/_libs/algos.pyx`
- `pandas/setup.py`

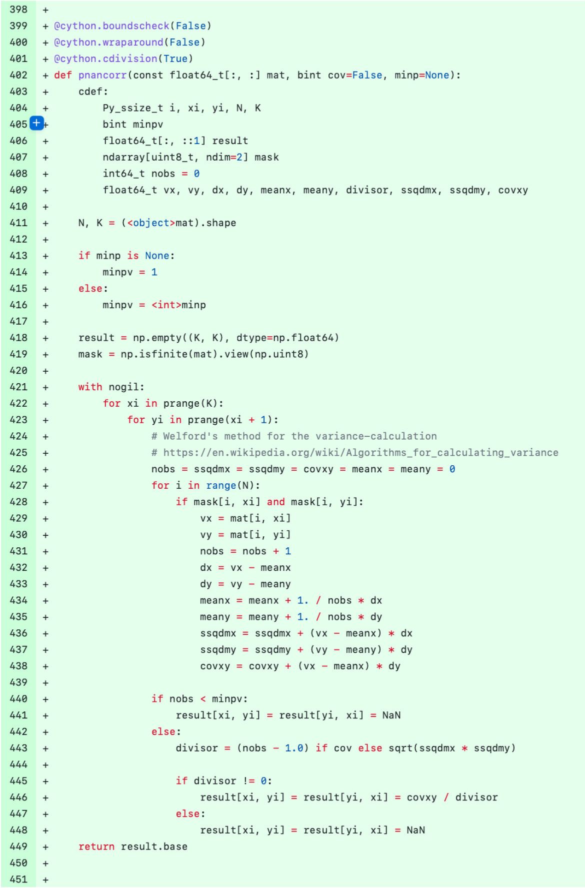
Files Modified

- `pandas/pandas/core/frame.py`

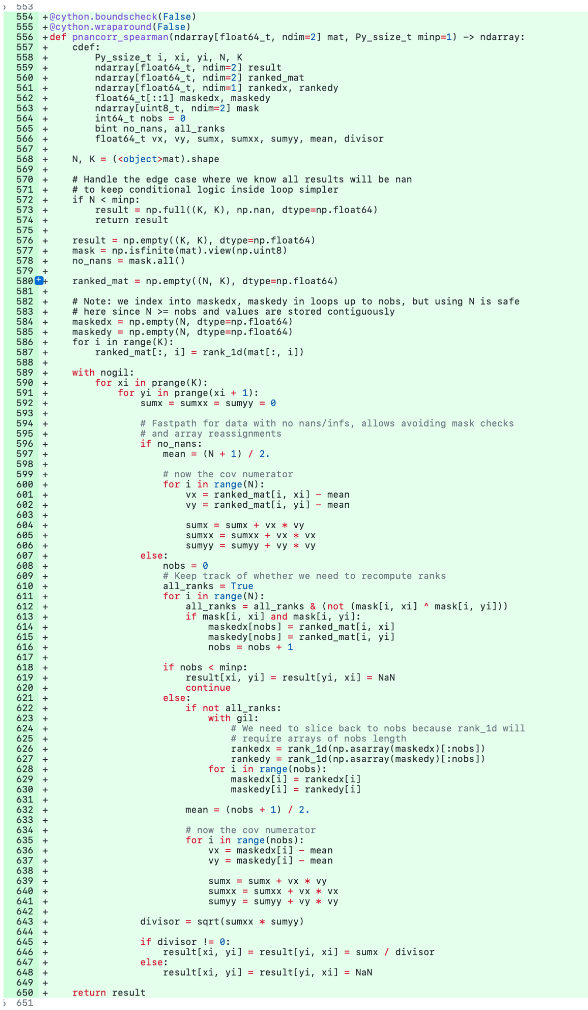


```diff
@@ -9954,6 +9954,7 @@ def corr(
         method: CorrelationMethod = "pearson",
         min_periods: int = 1,
         numeric_only: bool = False,
+        parallelize: bool = False,
     ) -> DataFrame:
         """
         Compute pairwise correlation of columns, excluding NA/null values.
@@ -10026,9 +10027,15 @@ def corr(
         mat = data.to_numpy(dtype=float, na_value=np.nan, copy=False)
 
         if method == "pearson":
-            correl = libalgos.nancorr(mat, minp=min_periods)
+            if parallelize:
+                correl = libalgos.pnancorr(mat, minp=min_periods)
+            else:
+                correl = libalgos.nancorr(mat, minp=min_periods)
         elif method == "spearman":
-            correl = libalgos.nancorr_spearman(mat, minp=min_periods)
+            if parallelize:
+                correl = libalgos.pnancorr_spearman(mat, minp=min_periods)
+            else:
+                correl = libalgos.nancorr_spearman(mat, minp=min_periods)
         elif method == "kendall" or callable(method):
             if min_periods is None:
                 min_periods = 1
```

- `pandas/pandas/_libs/algos.pyx`

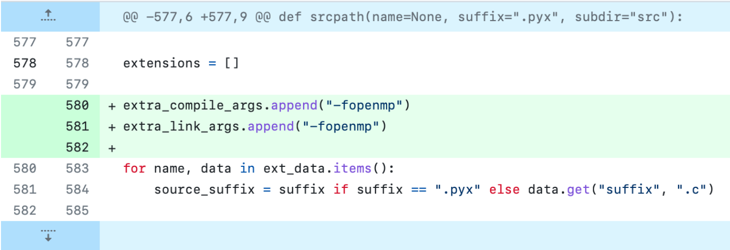


```
+
+ @cython.boundscheck(False)
+ @cython.wraparound(False)
+ @cython.cdivision(True)
+ def pnancorr(const float64_t[:, :] mat, bint cov=False, minp=None):
+     cdef:
+         Py_ssize_t i, xi, yi, N, K
+         bint minpv
+         float64_t[:, ::1] result
+         ndarray[uint8_t, ndim=2] mask
+         int64_t nobs = 0
+         float64_t vx, vy, dx, dy, meanx, meany, divisor, ssqdmx, ssqdmy, covxy
+
+     N, K = (<object>mat).shape
+
+     if minp is None:
+         minpv = 1
+     else:
+         minpv = <int>minp
+
+     result = np.empty((K, K), dtype=np.float64)
+     mask = np.isfinite(mat).view(np.uint8)
+
+     with nogil:
+         for xi in prange(K):
+             for yi in prange(xi + 1):
+                 # Welford's method for the variance-calculation
+                 # https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance
+                 nobs = ssqdmx = ssqdmy = covxy = meanx = meany = 0
+                 for i in range(N):
+                     if mask[i, xi] and mask[i, yi]:
+                         vx = mat[i, xi]
+                         vy = mat[i, yi]
+                         nobs = nobs + 1
+                         dx = vx - meanx
+                         dy = vy - meany
+                         meanx = meanx + 1. / nobs * dx
+                         meany = meany + 1. / nobs * dy
+                         ssqdmx = ssqdmx + (vx - meanx) * dx
+                         ssqdmy = ssqdmy + (vy - meany) * dy
+                         covxy = covxy + (vx - meanx) * dy
+
+                 if nobs < minpv:
+                     result[xi, yi] = result[yi, xi] = NaN
+                 else:
+                     divisor = (nobs - 1.0) if cov else sqrt(ssqdmx * ssqdmy)
+
+                     if divisor != 0:
+                         result[xi, yi] = result[yi, xi] = covxy / divisor
+                     else:
+                         result[xi, yi] = result[yi, xi] = NaN
+     return result.base
+
+
```



```
+@cython.boundscheck(False)
+@cython.wraparound(False)
+def pnancorr_spearman(ndarray[float64_t, ndim=2] mat, Py_ssize_t minp=1) -> ndarray:
+    cdef:
+        Py_ssize_t i, xi, yi, N, K
+        ndarray[float64_t, ndim=2] result
+        ndarray[float64_t, ndim=2] ranked_mat
+        ndarray[float64_t, ndim=1] rankedx, rankedy
+        float64_t[::1] maskedx, maskedy
+        ndarray[uint8_t, ndim=2] mask
+        int64_t nobs = 0
+        bint no_nans, all_ranks
+        float64_t vx, vy, sumx, sumxx, sumyy, mean, divisor
+
+    N, K = (<object>mat).shape
+
+    # Handle the edge case where we know all results will be nan
+    # to keep conditional logic inside loop simpler
+    if N < minp:
+        result = np.full((K, K), np.nan, dtype=np.float64)
+        return result
+
+    result = np.empty((K, K), dtype=np.float64)
+    mask = np.isfinite(mat).view(np.uint8)
+    no_nans = mask.all()
+
+    ranked_mat = np.empty((N, K), dtype=np.float64)
+
+    # Note: we index into maskedx, maskedy in loops up to nobs, but using N is safe
+    # here since N >= nobs and values are stored contiguously
+    maskedx = np.empty(N, dtype=np.float64)
+    maskedy = np.empty(N, dtype=np.float64)
+    for i in range(K):
+        ranked_mat[:, i] = rank_1d(mat[:, i])
+
+    with nogil:
+        for xi in prange(K):
+            for yi in prange(xi + 1):
+                sumx = sumxx = sumyy = 0
+
+                # Fastpath for data with no nans/infs, allows avoiding mask checks
+                # and array reassignments
+                if no_nans:
+                    mean = (N + 1) / 2.
+
+                    # now the cov numerator
+                    for i in range(N):
+                        vx = ranked_mat[i, xi] - mean
+                        vy = ranked_mat[i, yi] - mean
+
+                        sumx = sumx + vx * vy
+                        sumxx = sumxx + vx * vx
+                        sumyy = sumyy + vy * vy
+                else:
+                    nobs = 0
+                    # Keep track of whether we need to recompute ranks
+                    all_ranks = True
+                    for i in range(N):
+                        all_ranks = all_ranks & (not (mask[i, xi] ^ mask[i, yi]))
+                        if mask[i, xi] and mask[i, yi]:
+                            maskedx[nobs] = ranked_mat[i, xi]
+                            maskedy[nobs] = ranked_mat[i, yi]
+                            nobs = nobs + 1
+
+                    if nobs < minp:
+                        result[xi, yi] = result[yi, xi] = NaN
+                        continue
+                    else:
+                        if not all_ranks:
+                            with gil:
+                                # We need to slice back to nobs because rank_1d will
+                                # require arrays of nobs length
+                                rankedx = rank_1d(np.asarray(maskedx)[:nobs])
+                                rankedy = rank_1d(np.asarray(maskedy)[:nobs])
+                            for i in range(nobs):
+                                maskedx[i] = rankedx[i]
+                                maskedy[i] = rankedy[i]
+
+                        mean = (nobs + 1) / 2.
+
+                        # now the cov numerator
+                        for i in range(nobs):
+                            vx = maskedx[i] - mean
+                            vy = maskedy[i] - mean
+
+                            sumx = sumx + vx * vy
+                            sumxx = sumxx + vx * vx
+                            sumyy = sumyy + vy * vy
+
+                divisor = sqrt(sumxx * sumyy)
+
+                if divisor != 0:
+                    result[xi, yi] = result[yi, xi] = sumx / divisor
+                else:
+                    result[xi, yi] = result[yi, xi] = NaN
+
+    return result
```

- `pandas/setup.py`



```
@@ -577,6 +577,9 @@ def srcpath(name=None, suffix=".pyx", subdir="src"):

 extensions = []

+extra_compile_args.append("-fopenmp")
+extra_link_args.append("-fopenmp")
+
 for name, data in ext_data.items():
     source_suffix = suffix if suffix == ".pyx" else data.get("suffix", ".c")

```

Testing Suite

Make sure you run the following in the development environment in order to test:

python setup.py build_ext --inplace -j4 --with-debugging-symbols

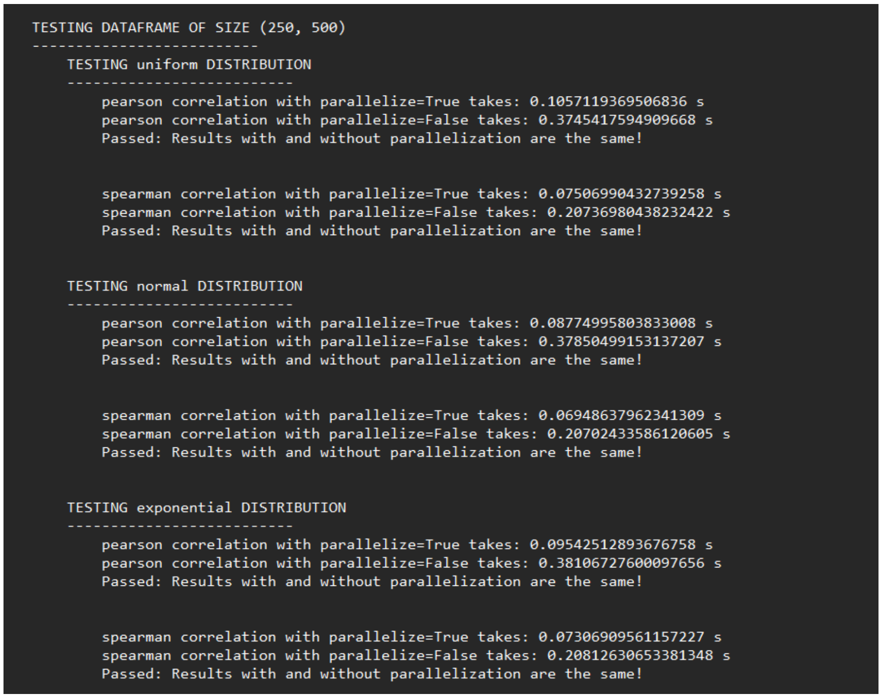
python -m pip install -e . --no-build-isolation --no-use-pep517

Acceptance Testing: found in `\pandas\pandas\` `ParallelAccTest.py`

First and foremost, we chose to test and compare our optimized algorithm on various data frame sizes and data distributions on both the pearson and spearman data correlations. We also ensured that the computation done both with and without parallelization produced the **same output.** The reason **why** these test cases were chosen to provide a comprehensive evaluation of the performance of our correlation calculations with and without parallelization on different sized data frames and distributions:

```
sizes = [(250, 500), (500, 1000), (750, 1500), (1000, 2000), (1250, 2500), (1500, 3000), (1750, 3500), (2000, 4000)]
distributions = {
    'uniform': lambda size: np.random.rand(*size),
    'normal': lambda size: np.random.normal(loc=0, scale=1, size=size),
    'exponential': lambda size: np.random.exponential(scale=1, size=size)
}
methods = ['pearson', 'spearman']
for size in sizes:
    print(f"    TESTING DATAFRAME OF SIZE {size}\n    --------------------------\n")
    for dist_name, dist_func in distributions.items():
        print(f"        TESTING {dist_name} DISTRIBUTION\n        --------------------------")
        data = dist_func(size)
        df = pd.DataFrame(data)
        for method in methods:
            corr_values = []
            for parallelize in [True, False]:
                start = time.time()
                corr = df.corr(method=method, parallelize=parallelize)
                end = time.time()
                corr_values.append(corr)
                print(f"            {method.upper()} correlation with parallelize={parallelize} takes:", end - start, "s")
            if  corr_values[0].equals(corr_values[1]):
                print("            SUCCESS: Results with and without parallelization are the same!")
            else: print("            FAILURE: Results with and without parallelization are not the same!")
            print("\n")
```

Sample Output from Testing Script:



```
TESTING DATAFRAME OF SIZE (250, 500)
--------------------------
    TESTING uniform DISTRIBUTION
    --------------------------
        pearson correlation with parallelize=True takes: 0.1057119369506836 s
        pearson correlation with parallelize=False takes: 0.3745417594909668 s
        Passed: Results with and without parallelization are the same!


        spearman correlation with parallelize=True takes: 0.07506990432739258 s
        spearman correlation with parallelize=False takes: 0.20736980438232422 s
        Passed: Results with and without parallelization are the same!


    TESTING normal DISTRIBUTION
    --------------------------
        pearson correlation with parallelize=True takes: 0.08774995803833008 s
        pearson correlation with parallelize=False takes: 0.37850499153137207 s
        Passed: Results with and without parallelization are the same!


        spearman correlation with parallelize=True takes: 0.06948637962341309 s
        spearman correlation with parallelize=False takes: 0.20702433586120605 s
        Passed: Results with and without parallelization are the same!


    TESTING exponential DISTRIBUTION
    --------------------------
        pearson correlation with parallelize=True takes: 0.09542512893676758 s
        pearson correlation with parallelize=False takes: 0.38106727600097656 s
        Passed: Results with and without parallelization are the same!


        spearman correlation with parallelize=True takes: 0.07306909561157227 s
        spearman correlation with parallelize=False takes: 0.20812630653381348 s
        Passed: Results with and without parallelization are the same!
```

FULL Acceptance Testing Usage and Results:

Usage: `python3` `test.py` `--cpu=X --short=T/F`

NOTE: It took approximately 30 minutes to run all of the tests you see below. For the convenience of the user/grader, we added a `-=short` flag to just run the same tests on sizes (250, 500), (1000, 2000), and (1500, 3000) which should only take approximately 5 minutes and shows some more dramatic output. That being, just one small sized case, one average sized case, and a large sized case.

We also added a `--cpu=X` flag to run the examples with X running cores. Keep in mind that if you choose X to be more cores than what your computer contains, it will simply just run the code with the max number of cores your CPU has.

DataFrame Size	**Distribution Type**	**Pearson Correlation**	**Spearman Correlation**
(250, 500)	Uniform	**with Parallel:** 0.1057119369506836s **w/o Parallel:** 0.3745417594909668s	**with Parallel:** 0.07506990432739258 s **without Parallel:** 0.20736980438232422 s
	Normal	**with Parallel:** 0.08774995803833008 s **without Parallel:** 0.37850499153137207 s	**with Parallel:** 0.06948637962341309 s **without Parallel:** 0.20702433586120605 s
	Exponential	**with Parallel:** 0.09542512893676758 s **without Parallel:** 0.38106727600097656 s	**with Parallel:** 0.07306909561157227 s **without Parallel:** 0.20812630653381348 s
(500, 1000)	Uniform	**with Parallel:** 0.7502007484436035 s **without Parallel:** 3.0338754653930664 s	**with Parallel:** 0.43984484672546387 s **without Parallel:** 1.525325059890747 s
	Normal	**with Parallel:** 0.8240785598754883 s **without Parallel:** 3.0674660205841064 s	**with Parallel:** 0.4314112663269043 s **without Parallel:** 1.5708751678466797 s
	Exponential	**with Parallel:** 0.8428099155426025 s **without Parallel:** 3.108384847640991 s	**with Parallel:** 0.44490814208984375 s **without Parallel:** 1.6381146907806396 s
(750,1500)	Uniform	**with Parallel:** 2.5104126930236816 s **without Parallel:** 10.634663343429565 s	**with Parallel:** 1.5163753032684326 s **without Parallel:** 5.703556060791016 s
	Normal	with Parallel: 2.5518956184387207 s without Parallel: 10.57186508178711 s	**with Parallel:** 1.5520858764648438 s **without Parallel:** 5.50699782371521 s
	Exponential	**with Parallel:** 2.5312509536743164 s **without Parallel:** 10.353914260864258 s	**with Parallel:** 1.3617911338806152 s **without Parallel:** 5.7780396938323975 s
(1000, 2000)	Uniform	**with Parallel:** 6.075455665588379 s **without Parallel:** 25.0881290435791 s	with Parallel: 3.420807361602783 s **without Parallel:** 13.819174766540527 s
	Normal	**with Parallel:** 5.8305768966674805 s **without Parallel:** 25.011849641799927 s	**with Parallel:** 3.190596103668213 s **without Parallel:** 12.606896877288818 s
	Exponential	**with Parallel:** 6.4168243408203125 s **without Parallel:** 25.019479751586914 s	**with Parallel:** 3.284809112548828 s **without Parallel:** 13.49602198600769 s
(1250, 2500)	Uniform	**with Parallel:** 12.5325608253479 s **without Parallel:** 49.39109444618225 s	**with Parallel**: 7.3284382820129395 s **without Parallel:** 28.876088857650757 s
	Normal	**with Parallel:** 13.046297311782837 s **without Parallel:** 47.589006662368774 s	**with Parallel:** 6.438707113265991 s **without Parallel:** 25.653429985046387 s
	Exponential	**with Parallel:** 11.529727458953857 s **without Parallel:** 47.95808815956116 s	**with Parallel:** 6.33501935005188 s **without Parallel:** 25.313028812408447 s
(1500, 3000)	Uniform	**with Parallel:** 20.021771907806396 s **without Parallel:** 81.5821042060852 s	**with Parallel:** 11.329249620437622 s **without Parallel:** 44.497355222702026 s
	Normal	**with Parallel:** 20.00104808807373 s **without Parallel:** 82.38195872306824 s	**with Parallel:** 10.76189136505127 s **without Parallel:** 46.79224681854248 s

	Exponential	**with Parallel:** 20.055269956588745 s **without Parallel:** 84.84153270721436 s	**with Parallel:** 12.931928396224976 s **without Parallel:** 45.84282326698303 s
(1750, 3500)	Uniform	**with Parallel:** 32.92101502418518 s **without Parallel:** 130.73699855804443 s	**with Parallel:** 18.50139808654785 s **without Parallel:** 75.55498433113098 s
	Normal	**with Parallel:** 31.994147300720215 s **without Parallel:** 130.30823683738708 s	**with Parallel:** 18.578843355178833 s **without Parallel:** 78.93502306938171 s
	Exponential	**with Parallel:** 32.73305821418762 s **without Parallel:** 132.0181427001953 s	**with Parallel:** 18.75952172279358 s **without Parallel:** 80.02709078788757 s
(2000, 4000)	Uniform	**with Parallel:** 49.373825788497925 s without Parallel: 197.9279670715332 s	**with Parallel:** 28.705546379089355 s **without Parallel:** 126.48924469947815 s
	Normal	**with Parallel:** 47.81682467460632 s **without Parallel:** 195.4403829574585 s	**with Parallel:** 27.650635242462158 s **without Parallel:** 122.64704465866089 s
	Exponential	**with Parallel:** 50.07123565673828 s **without Parallel:** 196.85616517066956 s	**with Parallel:** 30.50008988380432 s **without Parallel:** 124.75923418998718 s

Justification

Our tests show that parallelization generally improves the speed of the calculations. Specifically, for each data frame size and distribution tested, the time it takes to calculate the correlation using parallelization is shorter than without parallelization. The speedup seems to increase as the data frame size and the complexity of the distribution increase which is obvious.

Additionally, our tests show that the results obtained with and without parallelization are the same, indicating that parallelization does not affect the correctness of the calculations and simply just speeds it up.

Overall, these results suggest that parallelization can be a useful technique to speed up the calculation of correlations, especially for large data frames or complex distributions. However, the speedup obtained may depend on the specific hardware and software environment used for the calculations.

We also found that the computations produced the same output with and without parallelization. This further confirms that parallelization does not affect the correctness of our calculations.

Unit Testing: can be found in `pandas/pandas/parallelizeUnittests.py`

As for unit tests, we first ensured that the results being obtained by performing parallelism is the same as the results obtained without performing parallelism. This ensured consistency. We tested this on empty input and then input of large size.

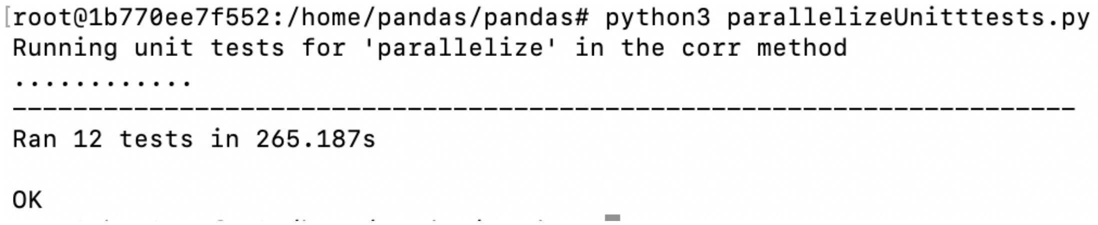
After confirming that the output was consistent, we then compared the execution times of the function with different settings of "parallelism." By doing so, you were able to confirm that the function indeed performs faster when "parallelism" is set to true and slower when it is set to false or not set at all.

Overall, our testing process involved running the function with different settings of "parallelism," comparing the outputs, and measuring the execution times to determine the performance of the function with different settings. This approach allowed us to identify the optimal setting of "parallelism" for the function and ensure that it is consistently providing the expected results.

```
import numpy as np
import pandas as pd
import time
import unittest
from pandas.testing import assert_frame_equal
```

```
class TestParallelizeCor(unittest.TestCase):
    print("Running unit tests for 'parallelize' in the corr method")

    ################################################################
    #These tests check if the results obtained when the parallelize
    #method being set to either True or False return the same results


    #Test with method "pearson" with empty pandas dataframe
    def testEqualityForPearson1(self):
        df = pd.DataFrame(np.random.rand(0, 0))
        pd_corr1 = df.corr(method='pearson', parallelize=False)
        pd_corr2 = df.corr(method='pearson', parallelize=True)
        assert_frame_equal(pd_corr1, pd_corr2)

    # Test with method "spearman" with empty pandas dataframe
    def testEqualityForSpearman1(self):
        df = pd.DataFrame(np.random.rand(0, 0))
        pd_corr1 = df.corr(method='spearman', parallelize=False)
        pd_corr2 = df.corr(method='spearman', parallelize=True)
        assert_frame_equal(pd_corr1, pd_corr2)

    #Test with method "pearson" with Pandas DataFrame with 1000
    #rows and 2000 columns, filled with random values between 0 and 1
    def testEqualityForPearson2(self):
        df = pd.DataFrame(np.random.rand(1000, 2000))
        pd_corr1 = df.corr(method='pearson', parallelize=False)
        pd_corr2 = df.corr(method='pearson', parallelize=True)
        assert_frame_equal(pd_corr1, pd_corr2)

    #Test with method "spearman" with Pandas DataFrame with 1000
    #rows and 2000 columns, filled with random values between 0 and 1
    def testEqualityForSpearman2(self):
        df = pd.DataFrame(np.random.rand(1000, 2000))
        pd_corr1 = df.corr(method='spearman', parallelize=False)
        pd_corr2 = df.corr(method='spearman', parallelize=True)
        assert_frame_equal(pd_corr1, pd_corr2)

    #Test with method "pearson" with Pandas DataFrame with 1000
    #rows and 2000 columns, filled with random values between 0 and 1
    #also set numeric_only parameter to True
    def testEqualityForPearson3(self):
        df = pd.DataFrame(np.random.rand(1000, 2000))
        pd_corr1 = df.corr(method='pearson', numeric_only=True, parallelize=False)
        pd_corr2 = df.corr(method='pearson', numeric_only = True, parallelize=True)
        assert_frame_equal(pd_corr1, pd_corr2)

    # Test with method "spearman" with Pandas DataFrame with 1000
    # rows and 2000 columns, filled with random values between 0 and 1
    # also set numeric_only parameter to True
    def testEqualityForSpearman3(self):
        df = pd.DataFrame(np.random.rand(1000, 2000))
        pd_corr1 = df.corr(method='spearman', numeric_only=True, parallelize=False)
        pd_corr2 = df.corr(method='spearman', numeric_only = True, parallelize=True)
        assert_frame_equal(pd_corr1, pd_corr2)

    # Test with method "pearson" with Pandas DataFrame with 1000
    # rows and 2000 columns, filled with random values between 0 and 1
    # also set min_periods parameter to 0#
    # This parameter is set to 1 by default
    def testEqualityForPearson4(self):
        df = pd.DataFrame(np.random.rand(1000, 2000))
        pd_corr1 = df.corr(method='pearson', min_periods=0, parallelize=False)
        pd_corr2 = df.corr(method='pearson', min_periods=0, parallelize=True)
        assert_frame_equal(pd_corr1, pd_corr2)

    # Test with method "spearman" with Pandas DataFrame with 1000
    # rows and 2000 columns, filled with random values between 0 and 1
    # also set min_periods parameter to 0#
    # This parameter is set to 1 by default
    def testEqualityForSpearman4(self):
        df = pd.DataFrame(np.random.rand(1000, 2000))
        pd_corr1 = df.corr(method='spearman', min_periods=0, parallelize=False)
        pd_corr2 = df.corr(method='spearman', min_periods=0, parallelize=True)
        assert_frame_equal(pd_corr1, pd_corr2)

    # Test with method "pearson" with Pandas DataFrame with 1000
    # rows and 2000 columns, filled with random values between 0 and 1
    # also set min_periods parameter to 10#
    # This parameter is set to 1 by default
    def testEqualityForPearson5(self):
        df = pd.DataFrame(np.random.rand(1000, 2000))
        pd_corr1 = df.corr(method='pearson', min_periods=10, parallelize=False)
        pd_corr2 = df.corr(method='pearson', min_periods=10, parallelize=True)
        assert_frame_equal(pd_corr1, pd_corr2)

    # Test with method "spearman" with Pandas DataFrame with 1000
    # rows and 2000 columns, filled with random values between 0 and 1
    # also set min_periods parameter to 10#
    # This parameter is set to 1 by default
    def testEqualityForSpearman5(self):
        df = pd.DataFrame(np.random.rand(1000, 2000))
        pd_corr1 = df.corr(method='spearman', min_periods=10, parallelize=False)
        pd_corr2 = df.corr(method='spearman', min_periods=10, parallelize=True)
        assert_frame_equal(pd_corr1, pd_corr2)

    # ###############################################################



    # ###############################################################
    # These tests check if the results obtained when the parallelize
    # method being set True returns a result faster than when it is
    # set to False

    # Test speed with method "pearson" with Pandas DataFrame with 1000
    # rows and 2000 columns, filled with random values between 0 and 1
    def testSpeedForPearson(self):
        df = pd.DataFrame(np.random.rand(1000, 2000))
        start = time.time()
        pd_corr = df.corr(method='pearson', parallelize=False)
        end1 = time.time() - start
        start = time.time()
        pd_corr = df.corr(method='pearson', parallelize=True)
        end2 = time.time() - start
        self.assertGreater(end1, end2)

    # Test speed with method "spearman" with Pandas DataFrame with 1000
    # rows and 2000 columns, filled with random values between 0 and 1
    def testSpeedForSpearman(self):
        df = pd.DataFrame(np.random.rand(1000, 2000))
        start = time.time()
        pd_corr = df.corr(method='spearman', parallelize=False)
        end1 = time.time() - start
        start = time.time()
        pd_corr = df.corr(method='spearman', parallelize=True)
        end2 = time.time() - start
        self.assertGreater(end1, end2)
    # ###############################################################

if __name__ == '__main__':
    unittest.main()
```

Terminal Output:



```
[root@1b770ee7f552:/home/pandas/pandas# python3 parallelizeUnitttests.py
Running unit tests for 'parallelize' in the corr method
............
----------------------------------------------------------------------
Ran 12 tests in 265.187s

OK
```