Issue #20595 Parallelization in Locally Linear Embedding

Link to Issue: https://github.com/scikit-learn/scikit-learn/issues/20595

Summary: The issue proposed a new feature that improves performance of a series of computations. As the issue suggests, we want to implement functionality that parallelizes the simple for loop used in barycenter_weights(). Specifically, the barycenter_weights() function takes some input which it uses in the simple for loop to compute various values and store those values into a nxn array that it then returns.

Implementation: We implemented a new function called barycenter_weights_parallel() that does the same thing as barycenter_weights() but the computation happens in parallel.

```
79  + def barycenter_weights_parallel(X, Y, indices, reg=1e-3):
```

To do this we imported a python library called multiprocessing (See https://docs.python.org/3/library/multiprocessing.html).

```
106 +     import multiprocessing
```

We redefined the nxn array to be a RawArray object so that it is thread safe (See https://docs.python.org/3/library/multiprocessing.html#multiprocessing.sharedctypes.RawArray).

```
115 +     B = multiprocessing.RawArray('d', n_samples * n_neighbors)
```

Instead of working with Process class and having to worry about concurrency, we decided we can implement our feature using a simpler abstraction of a Process offered in the multiprocessing library, called Pool.

```
132 +     with multiprocessing.Pool(multiprocessing.cpu_count()) as p:
133 +         p.map(parallel, enumerate(indices))
```

We defined a function within barycenter_weights_parallel() called parallel() that does the computation of the original simple for loop.

```
118 +     global parallel
119 +     def parallel(element):
120 +         A = Y[element[1]]
121 +         C = A - X[element[0]]  # broadcasting
122 +         G = np.dot(C, C.T)
123 +         trace = np.trace(G)
124 +         if trace > 0:
125 +             R = reg * trace
126 +         else:
127 +             R = reg
128 +         G.flat[:: n_neighbors + 1] += R
129 +         w = solve(G, v, sym_pos=True)
130 +         np.frombuffer(B).reshape((n_samples, n_neighbors))[element[0], :] = w / np.sum(w)
```

Now using parallel() and Pool we are able to execute the function in parallel on multiple threads and collect the results in our RawArray.

```
134 +
135 +     return np.frombuffer(B).reshape((n_samples, n_neighbors))
136 +
```

So now users can call barycenter_weights_parallel() to use the parallel version of the function.

There is another function called barycenter_kneighbors_graph() that calls barycenter_weights() in its execution. We've updated it to include an additional parameter called *parallel* with default value equal to True.

```
137 + def barycenter_kneighbors_graph(X, n_neighbors, reg=1e-3, n_jobs=None, parallel=True):
```

barycenter_kneighbors_graph() will now, in its execution, call either barycenter_weights() or barycenter_weights_parallel() depending on if *parallel* is set to True.

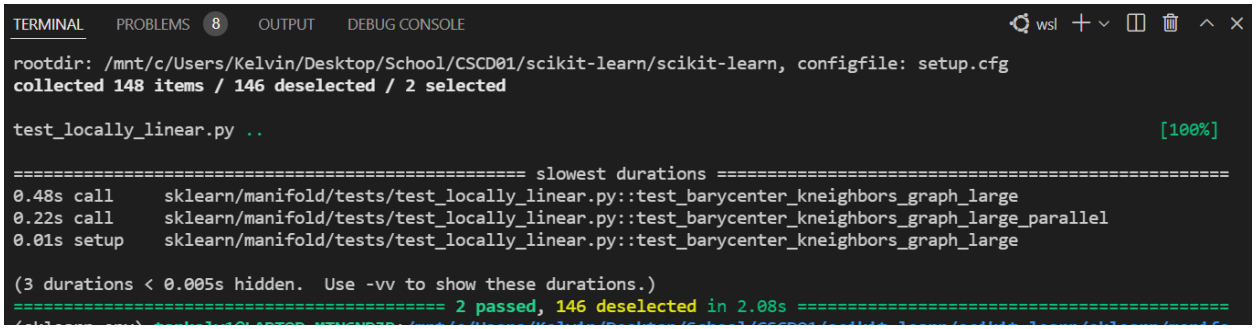
```
    -     data = barycenter_weights(X, X, ind, reg=reg)
174 +     if parallel:
175 +         data = barycenter_weights_parallel(X, X, ind, reg=reg)
176 +     else:
177 +         data = barycenter_weights(X, X, ind, reg=reg)
```

Acceptance Testing:

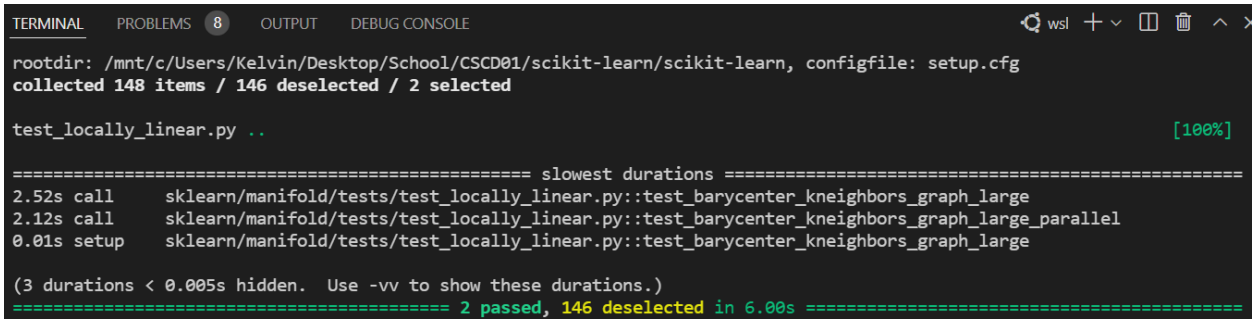
We updated test_locally_linear.py to include new test cases for matrices of different sizes 10x10, 100x100, 1000x1000, and 5000x5000 and for both the original and parallel implementations (see code). Although no improvements to minor deprovements were observed for small matrix inputs, the runtime did improve for larger matrix inputs. Below we have tests comparing the runtime for the original and parallel processes respectively (see pytests).

For Reference:

1000 x 1000 matrix inputs | 0.48s for sequential, 0.22s for parallel



3000 x 3000 matrix inputs | 2.52s for sequential, 2.12s for parallel



```
181    def test_barycenter_kneighbors_graph_large():
182        X = np.random.rand(1000, 1000)
183        A = barycenter_kneighbors_graph(X, 1, parallel=False)
184        A = barycenter_kneighbors_graph(X, 2, parallel=False)
185        pred = np.dot(A.toarray(), X)
186
187    def test_barycenter_kneighbors_graph_large_parallel():
188        X = np.random.rand(1000, 1000)
189        A = barycenter_kneighbors_graph(X, 1, parallel=True)
190        A = barycenter_kneighbors_graph(X, 2, parallel=True)
191        pred = np.dot(A.toarray(), X)
192
193    def test_barycenter_kneighbors_graph_xlarge():
194        X = np.random.rand(5000, 5000)
195        A = barycenter_kneighbors_graph(X, 1, parallel=False)
196        A = barycenter_kneighbors_graph(X, 2, parallel=False)
197        pred = np.dot(A.toarray(), X)
198
199    def test_barycenter_kneighbors_graph_xlarge_parallel():
200        X = np.random.rand(5000, 5000)
201        A = barycenter_kneighbors_graph(X, 1, parallel=True)
202        A = barycenter_kneighbors_graph(X, 2, parallel=True)
203        pred = np.dot(A.toarray(), X)
204
205    def test_barycenter_kneighbors_graph_medium():
206        X = np.random.rand(100, 100)
207        A = barycenter_kneighbors_graph(X, 1, parallel=False)
208        A = barycenter_kneighbors_graph(X, 2, parallel=False)
209        pred = np.dot(A.toarray(), X)
210
211    def test_barycenter_kneighbors_graph_medium_parallel():
212        X = np.random.rand(100, 100)
213        A = barycenter_kneighbors_graph(X, 1, parallel=True)
214        A = barycenter_kneighbors_graph(X, 2, parallel=True)
215        pred = np.dot(A.toarray(), X)
216
217    def test_barycenter_kneighbors_graph_small():
218        X = np.random.rand(10, 10)
219        A = barycenter_kneighbors_graph(X, 1, parallel=False)
220        A = barycenter_kneighbors_graph(X, 2, parallel=False)
221        pred = np.dot(A.toarray(), X)
222
223    def test_barycenter_kneighbors_graph_small_parallel():
224        X = np.random.rand(10, 10)
225        A = barycenter_kneighbors_graph(X, 1, parallel=True)
226        A = barycenter_kneighbors_graph(X, 2, parallel=True)
227        pred = np.dot(A.toarray(), X)
```

Unit Testing:

When using barycenter_weights function and specifying the *parallel* parameter as false, it should run the original implementation, otherwise the function should run the parallel implementation. To test for acceptance, call barycenter_weights() with *parallel* set to True. Running either implementation should produce the same results in the data.

Steps to reproduce:

1) Save the following code into a file called d.py:

```
import numpy as np
from sklearn.manifold._locally_linear import barycenter_kneighbors_graph
X = np.random.rand(1000, 1000)
A = barycenter_kneighbors_graph(X, 1, parallel=False)
A = barycenter_kneighbors_graph(X, 2, parallel=False)
B = barycenter_kneighbors_graph(X, 1, parallel=True)
B = barycenter_kneighbors_graph(X, 2, parallel=True)
print(np.array_equal(A.toarray(), B.toarray()))
```

2) Run d.py and see that A and B are equal