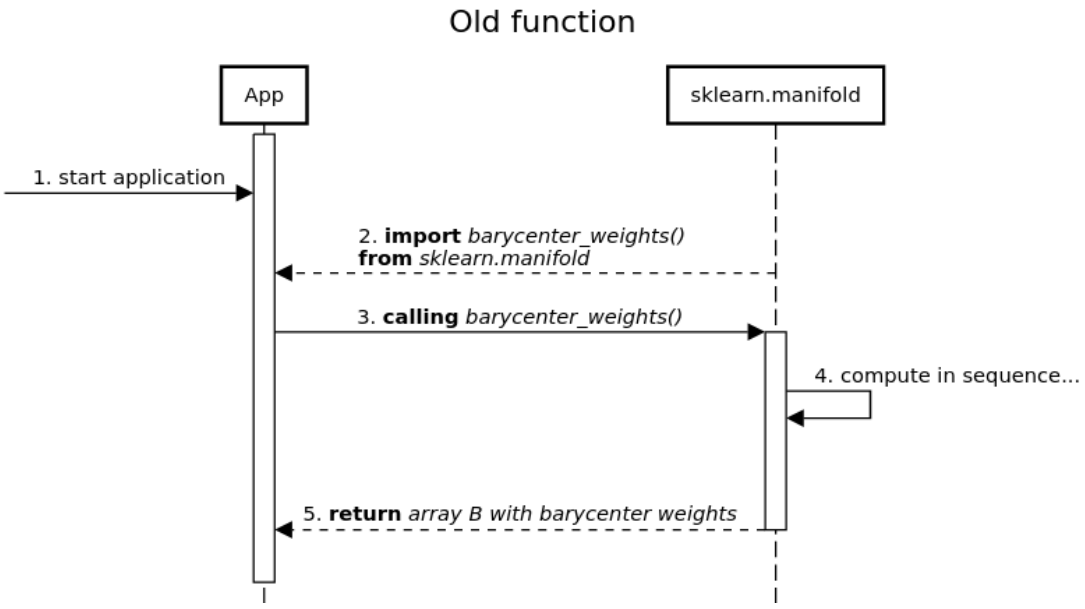
A4 Design Document

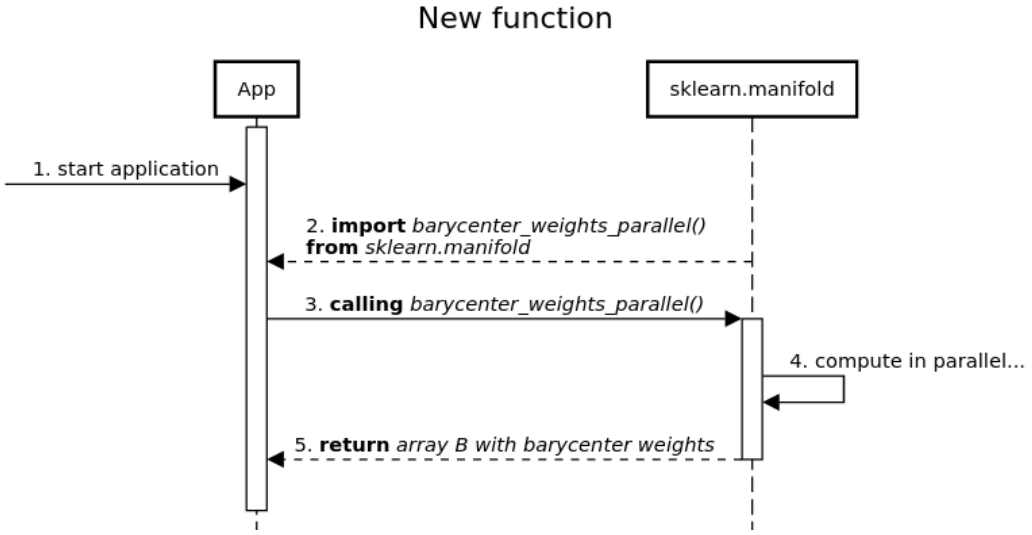
Keycap Guardians

Understanding the Issue

To approach this issue, we need to understand how *barycenter_weights()* works. So first, we came up with two sequence diagrams detailing how an application would interact with *barycenter_weights()* before and after our implementation.



Currently, we will calculate the barycenter weights using *barycenter_weights()*, which computes the barycenter weights sequentially.



We will implement a new function, *barycenter_weights_parallel()*, which will parallelize the expensive computation process by computing barycenter weights using multithreading techniques, utilizing the full computational power of multi-core CPUs, which will meaningfully reduce computation time for larger arrays.

Design Changes

File changed: *scikit-learn/sklearn/manifold/_locally_linear.py*

```
def barycenter_weights_parallel(X, Y, indices, reg=1e-3):
    """Compute barycenter weights of X from Y along the first axis

    We estimate the weights to assign to each point in Y[indices] to recover
    the point X[i]. The barycenter weights sum to 1.

    Parameters
    ----------
    X : array-like, shape (n_samples, n_dim)

    Y : array-like, shape (n_samples, n_dim)

    indices : array-like, shape (n_samples, n_dim)
            Indices of the points in Y used to compute the barycenter

    reg : float, default=1e-3
        amount of regularization to add for the problem to be
        well-posed in the case of n_neighbors > n_dim

    Returns
    -------
    B : array-like, shape (n_samples, n_neighbors)

    Notes
    -----
    See developers note for more information.
    """
    import multiprocessing

    X = check_array(X, dtype=FLOAT_DTYPES)
    Y = check_array(Y, dtype=FLOAT_DTYPES)
    indices = check_array(indices, dtype=int)

    n_samples, n_neighbors = indices.shape
    assert X.shape[0] == n_samples

    B = multiprocessing.RawArray('d', n_samples * n_neighbors)
    v = np.ones(n_neighbors, dtype=X.dtype)

    global parallel
    def parallel(element):
        A = Y[element[1]]
        C = A - X[element[0]]  # broadcasting
        G = np.dot(C, C.T)
        trace = np.trace(G)
        if trace > 0:
            R = reg * trace
        else:
            R = reg
        G.flat[:: n_neighbors + 1] += R
        w = solve(G, v, sym_pos=True)
        np.frombuffer(B).reshape((n_samples, n_neighbors))[element[0], :] = w / np.sum(w)

    with multiprocessing.Pool(multiprocessing.cpu_count()) as p:
        p.map(parallel, enumerate(indices))

    return np.frombuffer(B).reshape((n_samples, n_neighbors))
```

We left the original *barycenter_weights()* function as is and added the *barycenter_weights_parallel()* function which in effect is the same as

barycenter_weights(), except that we parallelized the above computationally expensive step by utilizing python's multiprocessing package using the pool object which offloads the work to CPU cores parallelly depending on the number of cores your computer has.

```
def barycenter_kneighbors_graph(X, n_neighbors, reg=1e-3, n_jobs=None, parallel=True):
    """Computes the barycenter weighted graph of k-Neighbors for points in X

    Parameters
    ----------
    X : {array-like, NearestNeighbors}
        Sample data, shape = (n_samples, n_features), in the form of a
        numpy array or a NearestNeighbors object.

    n_neighbors : int
        Number of neighbors for each sample.

    reg : float, default=1e-3
        Amount of regularization when solving the least-squares
        problem. Only relevant if mode='barycenter'. If None, use the
        default.

    n_jobs : int or None, default=None
        The number of parallel jobs to run for neighbors search.
        ``None`` means 1 unless in a :obj:`joblib.parallel_backend` context.
        ``-1`` means using all processors. See :term:`Glossary <n_jobs>`
        for more details.

    parallel : bool, default=True
        To run the barycenter_weights function in parallel or not.

    Returns
    -------
    A : sparse matrix in CSR format, shape = [n_samples, n_samples]
        A[i, j] is assigned the weight of edge that connects i to j.

    See Also
    --------
    sklearn.neighbors.kneighbors_graph
    sklearn.neighbors.radius_neighbors_graph
    """
    knn = NearestNeighbors(n_neighbors=n_neighbors + 1, n_jobs=n_jobs).fit(X)
    X = knn._fit_X
    n_samples = knn.n_samples_fit_
    ind = knn.kneighbors(X, return_distance=False)[:, 1:]
    if parallel:
        data = barycenter_weights_parallel(X, X, ind, reg=reg)
    else:
        data = barycenter_weights(X, X, ind, reg=reg)
    indptr = np.arange(0, n_samples * n_neighbors + 1, n_neighbors)
    return csr_matrix((data.ravel(), ind.ravel(), indptr), shape=(n_samples, n_samples))
```

We added a *parallel* parameter to *barycenter_kneighbors_graph()* with the default value equal to True, which will call *barycenter_weights_parallel()* inside it by default. However, the user may choose to use the sequential version of the *barycenter_weights()* if they wish by setting *parallel* to false.