Sequential and Shared-Memory Parallel Algorithms for Partitioned Local Depths

Aditya Devarakonda* Grey Ballard*

Abstract

In this work, we design, analyze, and optimize sequential and shared-memory parallel algorithms for partitioned local depths (PaLD). Given a set of data points and pairwise distances, PaLD is a method for identifying strength of pairwise relationships based on relative distances, enabling the identification of strong ties within dense and sparse communities even if their sizes and within-community absolute distances vary greatly. We design two algorithmic variants that perform community structure analysis through triplet comparisons of pairwise distances. We present theoretical analyses of computation and communication costs and prove that the sequential algorithms are communication optimal, up to constant factors. We introduce performance optimization strategies that yield sequential speedups of up to 29× over a baseline sequential implementation and parallel speedups of up to 26.2× over optimized sequential implementations using up to 32 threads on an Intel multicore CPU.

1 Introduction.

Partitioned local depths (PaLD) is a method for revealing community structure in distance-based data [2]. Given pairwise distances (or dissimilarities) of a set of points, PaLD computes another pairwise measure called cohesion that measures closeness based on relative distances. By relying on relative distance, PaLD is able to use a universal threshold to distinguish between strong and weak ties without defining neighborhoods by a single number of neighborhoods, neighborhood size, or absolute distance threshold. In this way, PaLD can identify neighborhoods of varying size and density, making it useful for data where the relationships among points behave differently across the space.

The input to PaLD is a distance matrix, and the output is a cohesion matrix. As detailed in Section 2, computing cohesion requires determining the size of the local neighborhood of each pair of points and then computing contributions to cohesion values based on neighborhood sizes. In each case, the fundamental operation is a comparison of the pairwise distances among triplets of points. Given n points, this yields an arithmetic complexity of $O(n^3)$. The goal of this paper is to develop efficient sequential and shared-memory parallel algorithms for scaling PaLD to datasets of size up to $O(10^5)$, making it computationally feasible to analyze ones that fit in memory on a single server. Section 3 presents the structure of the PaLD computation and our two main algorithmic approaches, which we call pairwise and triplet, respectively. As an $O(n^3)$ computation, PaLD shares many similarities with dense matrix multiplication (GEMM), and our algorithmic design borrows from ideas of cache-efficient algorithms for GEMM [3, 9, 19]. For example, the basic computation is a comparison between distances of points x, y, z, which involves distance matrix entries d_{xy}, d_{yz}, and d_{xz} and has an access pattern similar to the fused multiply-adds (FMA) within GEMM. There are a few key differences between PaLD and GEMM. First, because of symmetric distances, the order of the points is irrelevant, so rather than requiring consideration of all n^3 possible values of x, y, z, we need consider only $\binom{n}{3} \approx n^3/6$ unique triplets. Second, while the memory access of distances is regular, the updates of the cohesion requires branching based on distance comparisons. Finally, the computation requires two passes because cohesion updates depend on the sizes of local neighborhoods. Each pass requires a varying mix of integer and floating point operations in addition to the branching. The pairwise and triplet approaches navigate a tradeoff between exploiting symmetry and achieving regular data access and parallelization.

In Section 4 we prove a lower bound on the cache efficiency of any PaLD algorithm, and we show that both of our algorithms achieve optimal cache performance, up to constant factors. By exploiting symmetry and applying cache blocking, we obtain data locality in cache and minimize the number of reads and writes of matrix values. Section 5 details our low-level optimizations of the two PaLD algorithms. We show that branch avoidance has the highest impact on sequential performance given the high cost of branch misprediction [11, 14, 15]. Along with other optimizations including cache blocking and vectorization, we show performance improvements over naive sequential code of up to 29×. In Section 6 we design, optimize, and evaluate OpenMP parallel versions

*Department of Computer Science, Wake Forest University.

of the two PaLD algorithms. We show that the pairwise algorithm enables regular data access patterns and loop-based parallelism that can largely avoid write conflicts. The triplet algorithm exploits more symmetry to reduce arithmetic operations but requires task-based parallelism due to more complicated data access patterns and write conflicts. We also apply Non-Uniform Memory Access (NUMA) optimizations when scaling across sockets. We achieve strong scaling speedups up to 26.2× for pairwise and 19× for triplet over their optimized sequential versions on 32 threads. Finally, we describe PaLD scaling on text analysis and network data in Section 7, demonstrating the utility of PaLD on larger datasets, and we show a speedup of 30.3× on a task with $n = 23133$ using 32 threads.

2 Background.

Given a set of points and a pairwise distance metric, partitioned local depth (PaLD) algorithms determine the pairwise cohesion between all pairs of points in a dataset [2]. Assuming that the dataset comprises sufficiently separated subsets, cohesion values are invariant to contraction and dilation of distances within subset distances. Community structure revealed by cohesion values capture the concept of near neighbors based on relative positioning, adapting to varying density. This approach is more flexible than standard cluster labeling or nearest neighbor approaches.

Density-based approaches (e.g. DBSCAN) [5, 6, 7] attempt to combine points into high- and low-density groups based on pairwise distances similar to PaLD. However, these density-based approaches typically use a global density thresholding parameter, fixed for all points, which must be tuned by the user to reflect locality and cluster size.

Approaches like k-nearest neighbor (KNN) [8] also attempt to group points via pairwise distance comparisons. Unlike PaLD, KNN-based approaches fix the neighborhood size to k nearest neighbors (in terms of absolute distances) for a chosen point. The tuning parameter, k, is often fixed for all points. In contrast, cohesion values from PaLD depend on relative distances among triplets of points, which can be more reliable than exact numerical distances, particularly for high dimensional, non-Euclidean data points. In addition, PaLD requires $O(n^3)$ operations to compute cohesion values without assumptions on underlying probability distribution or tuning parameters.

Given a set of points $\mathcal{S}$, the *local focus* of a pair of points $x, y \in \mathcal{S}$ is the set of all points within distance d_{xy} of either x or y, where d_{xy} is the distance between x and y: $\mathcal{U}_{xy} = \{z \in \mathcal{S} \mid d_{xz} \leq d_{xy} \textbf{ or } d_{yz} \leq d_{xy}\}$. We let $u_{xy} = |\mathcal{U}_{xy}|$ denote the size of the local focus.

The *local depth* of a point $x \in \mathcal{S}$ is the probability that, given a uniformly chosen random second point $Y \in \mathcal{S}$ and a random third point Z chosen uniformly from the local focus $\mathcal{U}_{xY}$, Z is closer to x than Y:

$$\ell_x = \Pr\left[d_{Zx} < d_{ZY} \mid Y \sim \mathbb{U}(\mathcal{S}\backslash\{x\}), Z \sim \mathbb{U}(\mathcal{U}_{xY})\right]. \tag{2.1}$$

The cohesion of a point z to another point x is a part of the local depth ℓ_x and is defined as

$$c_{xz} = \Pr\left[Z = z \textbf{ and } d_{Zx} < d_{ZY}\right]. \tag{2.2}$$

The random variables Y and Z in Eq. (2.2) are chosen from the same distributions as in Eq. (2.1); we drop the notation here and later. This implies that $\ell_x = \sum_{z\in\mathcal{S}} c_{xz}$, or that cohesion is partitioned local depth. The cohesion matrix, C, can be used to analyze community structure. For example, two points have particularly strong cohesion if the impact of one of the points to the other is more than that expected from a random focus point of another random point.

3 PaLD Algorithms Design.

In order to compute the cohesion of all pairs of points, we can again use the law of total probability to partition c_{xz} across all points $y \in \mathcal{S}$:

$$c_{xz} = \sum_{y\in\mathcal{S}} \Pr\left[Y = y \textbf{ and } Z = z \textbf{ and } d_{Zx} < d_{ZY}\right].$$

Using the law of conditional probability, this becomes

$$c_{xz} = \sum_{y\in\mathcal{S}} \Pr\left[d_{zx} < d_{zy} \mid Y = y, Z = z\right] \cdot \Pr\left[Z = z \mid Y = y\right] \cdot \Pr\left[Y = y\right]$$

which implies

$$c_{xz} = \sum_{y\in\mathcal{S}} \mathbb{I}_{d_{xz}\leq d_{yz}} \cdot \frac{\mathbb{I}_{d_{xz}\leq d_{xy}}}{u_{xy}} \cdot \frac{1}{n-1} = \frac{1}{n-1} \sum_{y\in\mathcal{S}} g_{xyz}, \tag{3.3}$$

where $\mathbb{I}$ is the indicator function and we have defined

$$g_{xyz} = \mathbb{I}_{d_{xz}\leq d_{yz}} \cdot \mathbb{I}_{d_{xz}\leq d_{xy}} \,/\, u_{xy}. \tag{3.4}$$

The task is then to compute g_{xyz} for all $x, y, z \in \mathcal{S}$, a total of n^3 values. However, only approximately 1/3rd of the g_{xyz} values are nonzero because, given three points with unique pairwise distance values, only one pair has the minimum distance. For example, given points $x, y, z \in \mathcal{S}$, if x and y are the closest pair, then g_{xzy} and g_{yzx} are nonzero, but $g_{xyz} = g_{yxz} = g_{zxy} = g_{zyz} = 0$. To compute the nonzero values g_{xzy} and g_{yzx}, we need the values u_{xz} and u_{yz}. The size

Algorithm 1 Pairwise Sequential Algorithm

Require: $D \in \mathbb{R}^{n \times n}$, Distance Matrix
Ensure: $C \in \mathbb{R}^{n \times n}$, Cohesion Matrix

```
for x = 1 to n − 1 do
    for y = x + 1 to n do
        u_xy = 0
        for z = 1 to n do
            if d_xz < d_xy or d_yz < d_xy then
                u_xy = u_xy + 1
        for z = 1 to n do
            if d_xz < d_xy or d_yz < d_xy then
                if d_xz < d_yz then
                    c_xz = c_xz + 1/u_xy
                else
                    c_yz = c_yz + 1/u_xy
```

of any given local focus can be computed as $u_{xy} = \sum_{z \in \mathcal{S}} \mathbb{I}_{d_{xz} \le d_{xy} \text{ or } d_{yz} \le d_{xy}}$.

We consider two algorithmic approaches to computing the local focus sizes and the final cohesion matrix that take advantage of the symmetry. The first approach, which we call the *pairwise algorithm*, considers all $\binom{n}{2}$ pairs of points, and for each pair, first determines the size of its local focus and then computes contributions to the cohesion matrix from all points within the local focus. The second approach, which we call the *triplet algorithm*, considers all $\binom{n}{3}$ triplets of points, and for each triplet, determines which of the two local foci the triplet contributes to and then (in a second pass) determines which of the two cohesion matrix entries the triplet contributes to. We analyze and compare the two algorithms in Section 4.

3.1 Pairwise Algorithm. The entry-wise pairwise algorithm is given as Algorithm 1. The idea is to perform the computations for each pair of points x and y. To compute g_{xyz} for each third point z, we first must compute the size of the local focus, u_{xy}. This requires a pass over all n points with two comparisons and a possible integer increment. A second pass over all n points determines, for points in the local focus, which of the points x or y the third point supports, and the cohesion matrix is updated accordingly. Note that only one local focus size need be stored at any one time, requiring minimal temporary memory. Algorithm 1 was first proposed in [2] with an implementation in R. We use our C implementation of Algorithm 1 in our experiments as the baseline.

To improve the cache locality, we block the algorithm as follows: instead of considering only one pair of points, we consider two sets of points $\mathcal{X}$ and $\mathcal{Y}$ and consider all the pairs $(x, y) \in \mathcal{X} \times \mathcal{Y}$. In this way, we obtain locality on the distance matrix block $D_{\mathcal{X},\mathcal{Y}}$ and a temporary block of local focus sizes $U_{\mathcal{X},\mathcal{Y}}$.

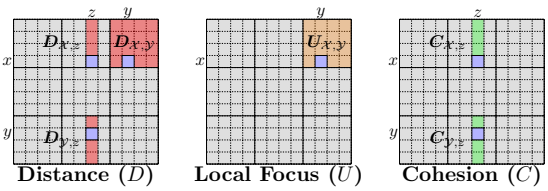

Figure 1: Dependency structure of the blocked pairwise algorithm. The highlighted regions represent quantities with temporal locality. Quantities in red correspond to reads and ones in green correspond to writes. Orange entries are computed and used in fast memory and then discarded. Blue represents entry-wise dependencies within each matrix/vector.

As in the entry-wise algorithm, the blocked algorithm makes two passes over all n third points. The first pass computes $U_{\mathcal{X},\mathcal{Y}}$, a local focus size block, and the second pass makes updates to the cohesion matrix.

Figure 1 shows the dependencies among the distance, local focus, and cohesion matrices for the blocked ($b = 4$) pairwise algorithm. The red blocks correspond to the entries of the distance matrix that are read and re-used while processing the pair of blocks $\mathcal{X}$ and $\mathcal{Y}$ (the pattern is the same in both passes, though $D_{\mathcal{X},\mathcal{Y}}$ remains in fast memory through both passes. The orange blocks represent entries of the local focus matrix which are computed in fast memory during the first pass and used during the second pass. The green blocks of the cohesion matrix are re-used during the second pass before being written back to slow memory. The blue blocks represent dependencies between entries of D, U, and C for one entry-wise iteration.

3.2 Triplet Algorithm. The entry-wise triplet algorithm is given as Algorithm 2. In Algorithm 1, if a third point z is in the local focus of x and y and is closer to x, then only the support of z for x is recorded in C (c_{xz} is updated). If z is closer to x in a focus with y, then x is closer to z in its focus with y. The idea of the triplet algorithm is to minimize the number of distance comparisons. By performing all the updates for each triplet of points, we can avoid redundant comparisons. However, this method requires that the local focus sizes are pre-computed for all pairs of points within the triplet, so it requires more temporary memory.

We can also block the triplet algorithm to obtain better cache locality. Instead of a single triplet of points, we consider three blocks $\mathcal{X}, \mathcal{Y}, \mathcal{Z}$ and all triplets $(x, y, z) \in \mathcal{X} \times \mathcal{Y} \times \mathcal{Z}$. We obtain locality on cache blocks of all three matrices: distance, local focus, and cohesion. Note that a first pass is required to compute the local focus matrix in its entirety, and then blocks

Algorithm 2 Triplet Sequential Algorithm.

Require: $D \in \mathbb{R}^{n \times n}$ Distance Matrix
Ensure: $C \in \mathbb{R}^{n \times n}$ Cohesion Matrix
Initialize $U = \texttt{triu}(2 * \texttt{ones}(n), 1)$
for $x = 1$ **to** $n - 1$ **do**
  for $y = x + 1$ **to** n **do**
    for $z = y + 1$ **to** n **do**
      if $d_{xy} < d_{xz}$ **and** $d_{xy} < d_{yz}$ **then**
        // x, y is closest pair
        $u_{xz} = u_{xz} + 1$
        $u_{yz} = u_{yz} + 1$
      else if $d_{xz} < d_{yz}$ **then**
        // x, z is closest pair
        $u_{xy} = u_{xy} + 1$
        $u_{yz} = u_{yz} + 1$
      else
        // y, z is closest pair
        $u_{xy} = u_{xy} + 1$
        $u_{xz} = u_{xz} + 1$
for $x = 1$ **to** $n - 1$ **do**
  for $y = x + 1$ **to** n **do**
    for $z = y + 1$ **to** n **do**
      if $d_{xy} < d_{xz}$ **and** $d_{xy} < d_{yz}$ **then**
        $c_{xy} = c_{xy} + 1/u_{xz}$
        $c_{yx} = c_{yx} + 1/u_{yz}$
      else if $d_{xz} < d_{yz}$ **then**
        $c_{xz} = c_{xz} + 1/u_{xy}$
        $c_{zx} = c_{zx} + 1/u_{yz}$
      else
        $c_{yz} = c_{yz} + 1/u_{xy}$
        $c_{zy} = c_{zy} + 1/u_{xz}$

of the local focus matrix are read from slow memory during the second pass as needed.

Figure 2 illustrates the dependencies among the distance, local focus, and cohesion matrices for the (blocked) triplet algorithm. In the first pass, the blocked triplet algorithm reads 3 blocks from the distance matrix, corresponding to the triplet pairs: $(x, y), (x, z), (y, z)$, and writes to the corresponding 3 blocks of the local focus matrix. Note that the distance and local focus matrices are symmetric so only the upper triangular parts are required. The cohesion matrix is not symmetric, thus in the second pass 6 blocks must be updated by performing distance comparisons (by reading $D_{\mathcal{X},\mathcal{Y}}, D_{\mathcal{X},\mathcal{Z}}, D_{\mathcal{Y},\mathcal{Z}}$) and utilizing entries of the local focus matrix (by reading $U_{\mathcal{X},\mathcal{Y}}, U_{\mathcal{X},\mathcal{Z}}, U_{\mathcal{Y},\mathcal{Z}}$).

4 Sequential Algorithm Analysis.

We model performance using the model, $\gamma F + \beta W$, where F and W represent an algorithm's computation and bandwidth costs, respectively, and γ (time per operation) and β (time per word moved) represent hardware parameters. We analyze communication cost assuming a two-level memory hierarchy, which contains fast memory (cache) of size M words and slow memory (DRAM)

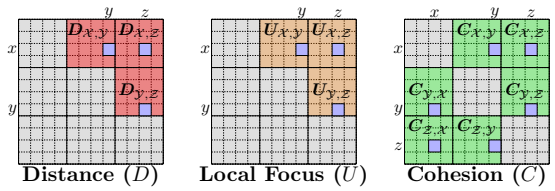

Figure 2: Dependency structure of the blocked triplet algorithm. The highlighted regions represent entries with temporal locality. Matrices in red correspond to reads and ones in green correspond to writes. Matrices in orange correspond to writes during the first pass and reads during the second pass. Blue represents the entry-wise dependencies within each matrix.

with unbounded size. We assume that computation can only be performed on operands residing in fast memory. If operands are in slow memory, then they must first be read into fast memory. We limit analysis in this section to a two-level memory hierarchy, but this memory model can be used to analyze communication for each adjacent pair of levels in a multi-level memory hierarchy.

4.1 Communication Lower Bounds.

We use the framework in [1] to derive communication lower bounds. The lower bound of [1, Theorem 2.6] applies to all three-nested-loops (3NL) computations as defined in that paper. We reproduce the 3NL definition here using the same notation, with sets $S_a, S_b, S_c \subseteq [n] \times [n]$ where $[n] = \{1, 2, \ldots, n\}$ and mappings $\mathbf{a} : S_a \to \mathcal{M}$, $\mathbf{b} : S_b \to \mathcal{M}$, $\mathbf{c} : S_c \to \mathcal{M}$, where $\mathcal{M}$ is slow memory. For each $(i, j) \in S_c$, we also have a set $S_{ij} \subseteq [n]$.

Definition 1. ([1, Definition 2.4]) *A computation is considered to be three-nested-loops (3NL) if it includes computing, for all $(i, j) \in S_c$ with S_{ij},*

$$Mem(\mathbf{c}(i, j)) = f_{ij}(\{g_{ijk}(Mem(\mathbf{a}(i, k)), Mem(\mathbf{b}(k, j)))\}_{k \in S_{ij}}),$$

where (a) mappings $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$ are all one-to-one into slow memory, and (b) functions f_{ij} and g_{ijk} depend nontrivially on their arguments.

We first verify that the cohesion matrix computation defined by Eqs. (3.3) and (3.4) is 3NL when the distance matrix is stored explicitly in memory. To satisfy the first constraint, we define the mappings $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ as all mapping onto the distance matrix (that is, each mapping is one-to-one but the three mappings are not disjoint). Here $\mathbf{a}(x, y)$ maps to the distance matrix entry d_{xy}. To satisfy the second constraint, we see that computing g_{xyz} depends nontrivially on $\mathbf{a}(x, y)$ and $\mathbf{b}(y, z)$, as both values must be compared with d_{xz} to evaluate the indicator functions, and computing c_{xz} depends nontrivially on its arguments, as it computes the

sum over all values. As argued in Section 3, the number of 3NL operations is $\sum_{i,j} |S_{ij}| = O(n^3)$. Then, by [1, Theorem 2.6], the bandwidth cost lower bound for PaLD is $W = \Omega(n^3/\sqrt{M})$.

4.2 Cost Analysis. The blocked algorithms are described in Section 3 with memory reference patterns depicted in Figs. 1 and 2. The loop structures of the blocked algorithms are shown (with OpenMP parallelization) in Figs. 5 and 7. We focus on the sequential costs in this section and discuss parallelization in Section 6. Since the algorithms require mixed comparison and arithmetic instructions, we explicitly define the hardware parameters γ_{cmp} and γ_{fma} to represent the time per instruction for floating-point comparisons and FMAs, respectively. We ignore the cost of integer arithmetic. Figure 5 shows the loop structure of the blocked pairwise algorithm where inner loop computations match Algorithm 1. We use b to represent the block size for the pairwise algorithm.

THEOREM 4.1. *The blocked pairwise algorithm has the leading order computation and communication costs:*

$$F = (5\gamma_{cmp} + 1\gamma_{fma}) \cdot n\binom{n}{2} \approx 3n^3 \text{ flops.}$$

$$W = 4\sqrt{2}\,\frac{n^3}{\sqrt{M}} \approx 5.7\,\frac{n^3}{\sqrt{M}} \text{ words moved.}$$

Proof. The blocked pairwise algorithm selects $\binom{n/b+1}{2}$ unique sets of points $\mathcal{X}, \mathcal{Y}$ with $|\mathcal{X}| = |\mathcal{Y}| = b$. A total of nb^2 iterations are required to determine if a third point, z, is in the local focus for each $(x, y) \in \mathcal{X} \times \mathcal{Y}$. The local focus update requires 2 floating-point comparisons followed by 1 integer accumulate into u_{xy}. The cohesion update requires 3 floating-point comparisons and 1 FMA, as the reciprocals of elements of $U_{\mathcal{X},\mathcal{Y}}$ can be precomputed once. When $\mathcal{X} = \mathcal{Y}$, only $n\binom{b}{2}$ iterations are required to perform local focus and cohesion updates. There are n/b such overlapping sets. Multiplying over the iterations, summing the work over the local focus and cohesion update loops, and multiplying by γ_{cmp} and γ_{fma} yields the computation cost.

Each of the $\binom{n/b+1}{2}$ possible combinations of $\mathcal{X} \times \mathcal{Y}$ points requires reading the $b \times b$ block $D_{\mathcal{X},\mathcal{Y}}$ from slow memory. In the first pass to compute the local focus sizes, for each third point, z, we read the two $b \times 1$ vectors $D_{\mathcal{X},z}$ and $D_{\mathcal{Y},z}$ from slow memory. The local focus block $U_{\mathcal{X},\mathcal{Y}}$ is computed and remains resident in fast memory. Similarly, each iteration of the second pass cohesion update requires reading the $b \times 1$ vectors $D_{\mathcal{X},z}, D_{\mathcal{Y},z}, C_{\mathcal{X},z}$ and $C_{\mathcal{Y},z}$ from slow memory. After each iteration within the second pass, $C_{\mathcal{X},z}$ and $C_{\mathcal{Y},z}$ must be written to slow memory. We must maintain $2b^2$ words of data in fast memory for $D_{\mathcal{X},\mathcal{Y}}$ and $U_{\mathcal{X},\mathcal{Y}}$, along with a constant number of length-b vectors, so $b \leq \sqrt{M/2}$ to leading order. Multiplying and summing these reads and writes over all iterations yields the leading order communication cost $4n^3/b$, and choosing $b \approx \sqrt{M/2}$ yields the result. ∎

Figure 7 shows the loop structure of the blocked triplet algorithm, and the inner loop computations match Algorithm 2. The local focus sizes and cohesion matrix updates are computed in two separate passes, and two block sizes $\hat{b}$ and $\tilde{b}$ can be tuned independently.

THEOREM 4.2. *The blocked triplet algorithm has the leading order computation and communication costs:*

$$F = (6\gamma_{cmp} + 2\gamma_{fma}) \cdot \binom{n}{3} \approx 1.33n^3 \text{ flops.}$$

$$W = \left(\sqrt{6} + 4\sqrt{3}\right)\frac{n^3}{\sqrt{M}} \approx 9.4\frac{n^3}{\sqrt{M}} \text{ words moved.}$$

Proof. The blocked local focus and cohesion matrix passes have the same loop structure, each selecting $\binom{n/b+2}{3}$ triplets of sets $\mathcal{X}, \mathcal{Y}$, and $\mathcal{Z}$ each of size b points, though the value of b differs in the two passes. The triplet algorithm contains 3 types of symmetry: $\mathcal{X} = \mathcal{Y} = \mathcal{Z}$, $\mathcal{X} \neq \mathcal{Y} = \mathcal{Z}$, and $\mathcal{X} = \mathcal{Y} \neq \mathcal{Z}$. While our implementation accounts for each type of symmetry, we ignore it in our leading order cost analysis. The local focus and cohesion update inner iterations each require 3 distance comparisons to determine the pair of points with minimum distance. The cohesion update iteration additionally requires 2 FMAs to update entries of the cohesion matrix. Multiplying operations by their respective γ terms and summing work over the two passes proves the computation cost.

There are $\binom{n/\hat{b}+2}{3}$ possible combinations of triplet blocks in the local focus pass. The local focus update must read 2 $\hat{b} \times \hat{b}$ blocks of D, read 2 $\hat{b} \times \hat{b}$ blocks of U, and write 2 $\hat{b} \times \hat{b}$ blocks of U from/to slow memory. Note that the block $D_{\mathcal{X},\mathcal{Y}}$ can be read and the block $U_{\mathcal{X},\mathcal{Y}}$ read and written only $\binom{n/\hat{b}+1}{2}$ times since they remain fixed while blocks $\mathcal{Z}$ vary in the innermost loop. The cohesion update requires reading 2 $\tilde{b} \times \tilde{b}$ blocks of D and U, respectively, followed by reading and writing 4 $\tilde{b} \times \tilde{b}$ blocks of C. The blocks $D_{\mathcal{X},\mathcal{Y}}$ and $U_{\mathcal{X},\mathcal{Y}}$ are read from slow memory and the blocks $C_{\mathcal{X},\mathcal{Y}}$ and $C_{\mathcal{Y},\mathcal{X}}$ can be read and written $\binom{n/\tilde{b}+1}{2}$ times. The total I/O cost is then $n^3/\hat{b} + 2n^3/\tilde{b}$, assuming that all blocks can be stored in fast memory. This requires that $\hat{b} \leq \sqrt{M/6}$ and $\tilde{b} \leq \sqrt{M/12}$ to leading order. Choosing block sizes at their approximate maximum value yields the communication cost. ∎

The constants for the communication cost in Theorem 4.2 can be improved by unblocking the innermost loop over $\mathcal{Z}$ for the local focus and cohesion update passes, which allows for a slightly larger block size. We use this technique for the pairwise algorithm, and it is useful in practice for matrix multiplication as well [19]. However, incorporating this optimization did not allow for auto-vectorization during cohesion updates where some updates require a stride of n. Blocking all three loops allowed for unit-stride for all cohesion updates. We provide more details in the following section.

We can conclude from Theorems 4.1 and 4.2 that the pairwise variant requires more computation than the triplet variant, but it moves less data. Both sequential variants attain the 3NL lower bound of $\Omega(n^3/\sqrt{M})$ and are communication-optimal within a constant factor. We will show in the next section how additional performance optimizations can yield large speedups. The optimized sequential algorithms serve as the baselines from which we derive efficient shared-memory parallel algorithms.

5 Sequential Performance Optimization.

We study the performance improvements achieved by each optimization, the tuning parameters introduced, and performance tradeoffs between the pairwise and triplet variants. All algorithms were written in C and compiled with the Intel C compiler (`icc`) release 2021.06. The code was compiled with the following compiler flags: `-Ofast -mavx512 -opt-zmm-usage=high`. Experiments are performed on a single-node, dual-socket platform with two Intel Xeon Gold 6226R CPUs (16 cores per socket). This CPU dynamically adjusts CPU frequency from 2.5 GHz (for 32 threads) to 3.9 GHz (for single-thread). The experiments and roofline analysis presented in this section use a CPU frequency of 3.9 GHz. We run 5 trials for each experiment and use the mean to compute speedups. We observe low runtime variance across trials, so we omit error bars for simplicity. We perform experiments on randomly generated distance matrices for powers of two $n \in \{128, \ldots, 4096\}$. Our code can handle arbitrary square matrix sizes, but we limit performance evaluation to powers of two.

5.1 Cache Blocking We begin performance tuning by applying one level of blocking to Algorithm 1 (naive pairwise) and Algorithm 2 (naive triplet). We show speedups relative to the naive implementation in Fig. 3 with a fixed $n = 2048$ matrix. Overall speedup over naive pairwise (resp. naive triplet) may be obtained by multiplying speedups across all optimizations. Naive triplet resulted in a speedup of 1.11× over naive pairwise due to less computation. Introducing one level of blocking to naive pairwise led to a speedup of 1.07×. Applying blocking to the triplet variant led to speedups of 1.20× over naive triplet (1.33× over naive pairwise).

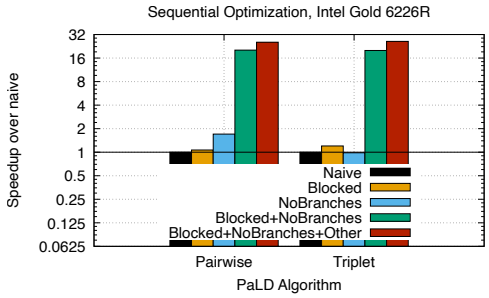

Figure 3: Speedup achieved from various performance optimizations applied to the Pairwise and Triplet algorithms. Speedups are arranged by algorithm and relative to the naive implementation.

5.2 Branch Avoidance Algorithms 1 and 2 require branches to correctly update U and C based on distance comparisons. Distance comparisons can be vectorized, but updates to U and C cannot due to branching. We avoid branches in both algorithms by computing auxiliary mask variables and performing FMAs with these explicit masks. Similar use of branch avoidance is shown to be effective in graph algorithms and improve performance despite performing extra computation [10]. For Algorithm 1, we compute the masks: $r = d_{xz} < d_{xy}$ **or** $d_{yz} < d_{xy}$ and $s = d_{xz} < d_{yz}$. The variable r indicates that z is in the (x, y) local focus and s determines the entry of C to update. C can be updated via two FMAs: $c_{xz} = c_{xz} + r \cdot s \cdot (1/u_{xy})$ and $c_{yz} = c_{yz} + (r)(1-s)(1/u_{xy})$. Assuming random labeling of points, $r = 1$ (c_{xz} or c_{yz} is updated) with probability $2/3$ and $s = 1$ (c_{xz} is updated) with probability $1/2$. Branch avoidance introduces a performance tradeoff by increasing computation (e.g. performing FMAs with explicit zeros) but eliminates branch misprediction overhead. For Algorithm 1, branch avoidance enables a fixed stride length for updates of C and facilitates compiler-aided optimizations, such as auto-vectorization and loop unrolling. Branch avoidance alone yielded a speedup of 1.7× over naive pairwise. While branch avoidance allows for vectorization, updates to c_{xz} and c_{yz} require a stride length of n. After blocking, we reduce the stride length to 1 by updating columns of C instead (see Fig. 1). The combination achieved speedups of 20.2× over naive pairwise.

Algorithm 2 must find the minimum distance among the (d_{xy}, d_{xz}, d_{yz}) triplet of pairwise distances. Assuming random labeling of points, each branch may be executed with a probability of $1/3$. As a result,

branch avoidance is also critical for Algorithm 2. We avoid branches by computing three masks from three floating point comparisons: $r = d_{xy} < d_{xz}$ **and** $d_{xy} < d_{yz}$, $s = (1-r)(d_{xz} < d_{yz})$, and $t = (1-r)(1-s)$. C can then be updated using six FMAs:

$$c_{xy} = c_{xy} + r\,(1/u_{xz})\,, \quad c_{yx} = c_{yx} + r\,(1/u_{yz})\,,$$
$$c_{xz} = c_{xz} + s\,(1/u_{xy})\,, \quad c_{zx} = c_{zx} + s\,(1/u_{yz})\,,$$
$$c_{yz} = c_{yz} + t\,(1/u_{xy})\,, \quad c_{zy} = c_{zy} + t\,(1/u_{xz})\,.$$

Applying branch avoidance to the triplet algorithm yields a speedup of 0.98× due to the stride-n updates to C. When combined with blocking, however, we attain speedups of 20× over naive triplet. Triplet with branch avoidance and blocking yields a speedup of 1.1× over pairwise with the same optimizations.

5.3 Other Optimizations We were able to extract additional speedup by replacing floating point operations with integer operations during local focus updates, and ignoring equality in pairwise/triplet distance comparisons. Each entry of U counts the number of points in the local focus based on distance comparisons, with results stored in a mask register. If U is stored as a floating point array, then each increment to update U requires an expensive integer mask to 32-bit floating point cast operation. We avoid this by storing U as an integer array during the local focus computation. This allowed us to combine casting with computing reciprocals prior to cohesion updates.

The theoretical formulation of PaLD [2] allows for ties in pairwise distances (e.g., $d_{xz} == d_{yz}$). When ties occur, support is split between cohesion entries c_{xz} and c_{yz} (i.e. $c_{xz} = c_{xz} + r \cdot s \cdot (0.5/u_{xy})$). In finite arithmetic, floating point equality is unlikely due to round-off and truncation. Avoiding ties is critical for Algorithm 2 which contains more distance tie permutations than pairwise. Introducing these additional optimizations yields self-relative speedups (over naive) of 25.5× and 26.2× for pairwise and triplet, respectively. Overall, optimized triplet achieves a speedup of 1.14× over optimized pairwise for $n = 2048$.

We also perform block size tuning for each algorithm. We experiment with (powers of two) block sizes in the range $[2^5, 2^{10}]$. Optimized pairwise attains a maximum speedup of 25.5× for $n = 2048$ after tuning.

For optimized triplet, updates to U require storing 3 distinct blocks of D and 3 distinct blocks of U in cache. Updates to C require 3 distinct blocks of D, 3 distinct blocks of U, and 6 distinct blocks of C in cache. This suggests that different block sizes may be better than a fixed block size. Figure 4 (bottom) illustrates the speedups observed (over Algorithm 2) for various block size combinations for the optimized triplet algorithm.

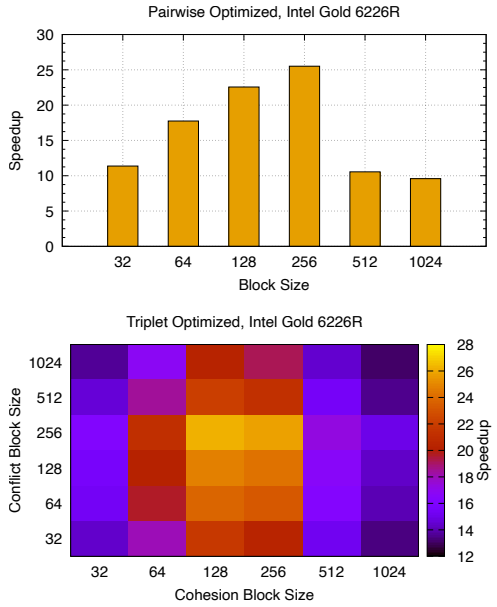

Figure 4: Speedup achieved from block size tuning for pairwise (top) and triplet (bottom) for $n = 2048$.

n	Pairwise Optimized	Triplet Optimized
128	**0.00117 (1.58×)**	0.00185
256	**0.00497 (1.34×)**	0.00665
512	**0.0188 (1.18×)**	0.0221
1024	0.1274	**0.1208 (1.05×)**
2048	0.9942	**0.8734 (1.14×)**
4096	8.3623	**6.6111 (1.26×)**

Table 1: Running time in seconds (and speedup) comparison of pairwise and triplet algorithms.

We observe a maximum speedup of 26.2× over naive triplet with $\hat{b} = 256$ and $\tilde{b} = 128$. In Table 1 we compare running times (and speedups) of optimized pairwise and optimized triplet over a range of input matrix sizes. For small matrix sizes, where D, U and C all fit in cache, optimized pairwise is fastest (e.g. speedup of 1.58× over triplet at $n = 128$). This is because n/b is a small integer where lower order terms dominate (see Theorem 4.2). For larger matrices, optimized triplet performs better (speedup of 1.26× over pairwise at $n = 4096$) due to lower computation cost. In practice, we expect triplet to be the better sequential variant for most applications of PaLD. If distances ties must be handled correctly, then pairwise is the better variant due to fewer branches.

The combination of all optimizations achieves speedups of 25.5× and 29× for pairwise and triplet, respectively, over naive pairwise (for $n = 2048$). We observe speedups of 23× and 26.2× over naive triplet.

5.4 Roofline Analysis We show in this section that the optimized pairwise implementation attains 27.7% of

hardware peak at $n = 2048$ and optimized triplet attains 28% at $n = 8192$. Our Intel CPU has a single-core, single precision peak of 249.6 Gflops/sec. Single precision comparisons on our CPU have a cycles-per-instruction (CPI) of 1 while all other single precision ops have a CPI of 0.5. Thus, floating point comparisons are twice as expensive. The optimized sequential pairwise algorithm requires 2 comparisons during local focus update to determine if a point, z, is in the neighborhood. The local focus matrix is incremented based on these comparisons. However, since U is stored in integer format, we ignore the cost of integer increments during the local focus pass. The cohesion update requires 3 comparisons: 2 comparisons to compute mask r which determines if a point z is in the local focus and 1 comparison to compute mask s which determines the column entry of C to update. Since results of floating point comparisons are stored in unsigned (integer) format, r and s must be cast to 32-bit floats before FMAs. This requires 2 unsigned int to floating point cast operations. Finally, 2 FMAs (each FMA requires two instructions) can be used to update c_{xz} and c_{yz}. We explicitly compute both entries and accumulate with an explicit zero, as this avoids branching. The total number of operations for sequential pairwise can be computed as follows:

$$F = (5\gamma_{cmp} + 2 \cdot 2\gamma_{fma} + 2\gamma_{cast}) \cdot n\binom{n}{2}$$

On our Intel Xeon Gold 6226R CPU, floating point comparisons have a CPI of 1 whereas FMAs and casting each have a CPI of 0.5. Since comparisons are twice as expensive, we normalize our operation count to be relative to FMA/cast. After normalization, the total number of operations become:

$$F = 16\gamma \cdot n\binom{n}{2} \approx 8n^3 \text{ ops.}$$

Finally, percentage of peak can be calculated by:

$$\frac{1}{249.6} \cdot \frac{F}{10^9 \cdot t_n} \tag{5.5}$$

where t_n is the runtime time (in seconds) obtained empirically from executing the optimized sequential pairwise algorithm on a matrix of size n and 249.6 Gflops/sec is the single precision, single core machine peak of our Intel CPU. The setting $n = 2048$ and $t_n = 0.994$ seconds (averaged over 5 trials) yields 27.7% of peak. The optimized sequential triplet algorithm makes two passes: one to compute U in its entirety and one to compute C. The triplet algorithm, which ignores ties, requires 6 comparisons across the two passes to uniquely determine the pair of points in a triplet with minimum pairwise distance. The local focus pass and cohesion pass must compute these distances. Once again, we ignore integer increments in the local focus pass. The remaining instructions are 3 casting and 6 FMA operations to update entries of C.

$$F = (12\gamma_{cmp} + 2 \cdot 6\gamma_{fma} + 3\gamma_{cast}) \cdot \binom{n}{3} \approx 6.5n^3.$$

Setting $F = 6.5n^3, n = 8192$ and $t_n = 51.16$ seconds in (5.5) yields 28% of peak. For reference, sequential SGEMM attains 80.5% of peak for $n = 2048$.

6 Shared-Memory Parallel Algorithms

This section presents the OpenMP parallelization of the optimized sequential pairwise and triplet algorithms. Figure 5 shows the OpenMP version of the blocked pairwise algorithm. The blocked pairwise algorithm first computes $U_{\mathcal{X},\mathcal{Y}}$ with a pass over all n points z. The local focus z-loop can be parallelized across p threads using the OpenMP `parallel for` construct. All threads must write to $U_{\mathcal{X},\mathcal{Y}}$ so a sum-reduction is required to resolve write conflicts. The cohesion update pass requires the quantities $1/u_{xy}\ \forall\ (x, y) \in \mathcal{X}\times\mathcal{Y}$, which can be parallelized without write conflicts. Cohesion updates are within each column of C to entries of $C_{\mathcal{X},z}$ and $C_{\mathcal{Y},z}$. The cohesion pass can be parallelized without write conflicts by splitting the z-loop across p threads. Figure 6 illustrates the write patterns for optimized OpenMP pairwise for $n = 16$, $b = 4$, and $p = 8$. Updates to entries of C requires corresponding entries from D, so D can also be partitioned column-wise. The pairwise algorithm is amenable to NUMA optimizations due to the regular data dependencies.

Figure 7 shows the OpenMP version of the blocked triplet algorithm. The triplet approach requires reading all of D for local focus and cohesion update passes. Blocking is performed over triplets of points, $\mathcal{X}, \mathcal{Y}, \mathcal{Z}$, and updates to U and C become irregular. We use the OpenMP tasking model [18] for parallelism. Each triplet block, $\mathcal{X} \times \mathcal{Y} \times \mathcal{Z}$, is a new task that can be executed by any available thread. Tasks in the local focus pass write to 3 blocks of U. C is not symmetric, so the cohesion update pass writes to 6 blocks. Write conflicts arise when multiple tasks need to update the same blocks of U or C. We resolve conflicts by annotating dependencies using the `depend` clause with the `inout` modifier. Figure 8 shows the write conflicts for the local focus pass. Each vertex represents one of the $\binom{n/b+2}{3}$ tasks and is labeled by $\mathcal{X}, \mathcal{Y}, \mathcal{Z}$ block values, and edges represent conflicts. The degree for each vertex varies based on the symmetry in the block. This leads to irregular dependencies which we will show in Section 6.1 are not as amenable to NUMA optimizations.

```
for(xb = 0; xb < n/b; ++xb)
  for(yb = 0; yb <= xb; ++yb)
    #pragma omp parallel for \
    reduction(+:U[𝒳,𝒴])
    for(z = 0; z < n; ++z)
      for(x = 0; x < b; ++x)
        y_start = (xb==yb) ? (x+1) : 0;
        for(y = y_start; y < b; ++y)
          // update u_xy.
    #pragma omp parallel for
    for(i = 0; i < b*b; ++i)
        U[i] = 1/U[i];
    #pragma omp parallel for
    for(z = 0; z < n; ++z)
      for(x = 0; x < b; ++x)
        y_start = (xb==jb) ? (x+1) : 0;
        for(y = y_start; y < b; ++y)
          // update c_xz and c_yz.
```

Figure 5: Blocked OpenMP pairwise Algorithm.

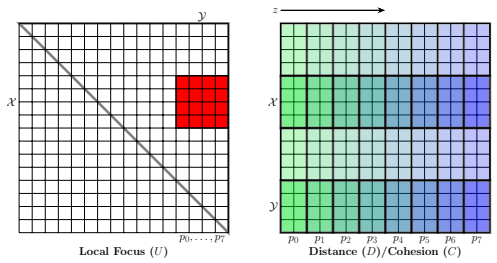

Figure 6: Distance matrix reads and Local Focus/Cohesion writes for parallel pairwise code with $n = 16$, $b = 4$, and $p = 8$. All threads have write conflicts to the U block for each pair $\mathcal{X}, \mathcal{Y}$ (in red), so synchronization is required via reductions. Only one U block is needed in fast memory at any given point in time. Writes to C are within one column, so column blocks can be partitioned across threads without write conflicts.

```
#pragma omp single
for(xb = 0; xb < n/b; ++xb)
  for(yb = xb; yb < n/b; ++yb)
    for(zb = yb; xb < n/b; ++zb)
      x_end=(xb==yb && yb==zb)?(b-1):b
      #pragma omp task untied depend(inout,
          U[𝒳,𝒴],U[𝒳,𝒵],U[𝒳,𝒵])
      for(x = 0; x < x_end; ++x)
        y_start=(xb==yb) ? (x+1) : 0;
        for(y = y_start; y < b; ++y)
          z_start=(yb==zb) ? (y+1) : 0;
          for(z = z_start; z < zb; ++z)
            // update u_xy, u_xz, u_yz.
#pragma omp parallel for
for(i = 0; i < n*n; ++i){
    U[i] = 1/U[i];
}
#pragma omp single
for(xb = 0; xb < n/b; ++xb)
  for(yb = xb; yb < n/b; ++yb)
    for(zb = zb; xb < n/b; ++zb)
      x_end=(xb==yb && yb==zb)?(b-1):b
      #pragma omp task untied depend(inout,
          C[𝒳,𝒴],C[𝒳,𝒵],C[𝒴,𝒵],
          C[𝒴,𝒳],C[𝒵,𝒳],C[𝒵,𝒴])
      for(x = 0; x < xend; ++x)
        y_start=(xb==yb) ? (i+1) : 0;
        for(y = ystart; y < b; ++y)
          z_start=(yb==zb) ? (y+1) : 0;
          for(z = z_start; z < zb; ++z)
            // update c_xy, c_xz, c_yz,
            // update c_yx, c_zx, c_zy.
```

Figure 7: Blocked OpenMP triplet Algorithm.

6.1 OpenMP Performance. We use OpenMP version 4.5 and test the OpenMP algorithms on randomly generated dense distance matrices with $n \in \{2048, 4096, 8192\}$. We incorporate NUMA optimizations into the pairwise algorithm by controlling thread affinity via the `OMP_PROC_BIND` and `OMP_PLACES` environment variables. We map OpenMP threads to physical cores, by assigning OpenMP thread ids 0 to 16 to CPU 0 and threads 17 to 31 to CPU 1. Finally, we disabled dynamic CPU frequency adjustment and fixed the frequency to 2.5 GHz, which corresponds to the max frequency at 32 threads, for experiments in this section.

A static loop schedule yields best performance due to the pairwise algorithm's regular dependencies. Each thread reads columns of D and C from thread-local fast memory so updates to C are spatially local. Thread binding ensures that accesses are temporally local by assigning fixed column blocks of D/C to threads. OpenMP allocates memory pages using a first-touch policy by default. If a single thread allocates D, then D resides in the memory hierarchy of the thread's CPU. D is typically computed outside the scope of the OpenMP algorithms, so we also study the effects of partitioning D across sockets (i.e. memory binding).

Figure 9 shows the speedup achieved by introducing thread binding only and thread + memory binding into the OpenMP pairwise algorithm across three matrix sizes, $n \in \{2048, 4096, 8192\}$. We use the OpenMP pairwise algorithm without NUMA-aware optimizations as our baseline and report speedups for 32 OpenMP threads. When we use thread binding only, we observe average speedups of $1.4\times, 1.5\times$, and $1.13\times$ for $n = 2048, 4098$, and 8192, respectively. Thread binding with memory binding yields average speedups of speedup of $1.7\times, 1.69\times$, and $1.2\times$ over the baseline. We did not perform TLB optimizations, therefore, we observe decreasing speedups for large matrix sizes. We also found that NUMA optimizations are useful at smaller thread counts, $2 \leq p \leq 16$, by mapping half the threads to CPU 0 and the other half to CPU 1. This map-

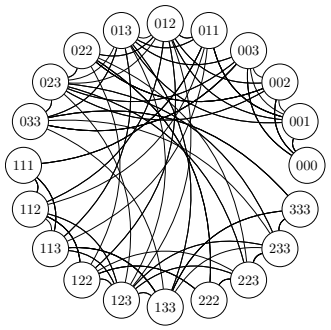

Figure 8: Task diagram for parallel triplet with $n/b = 4$, where nodes are labeled by their $\mathcal{X}, \mathcal{Y}, \mathcal{Z}$ block values. Edges represent write conflicts for U between tasks.

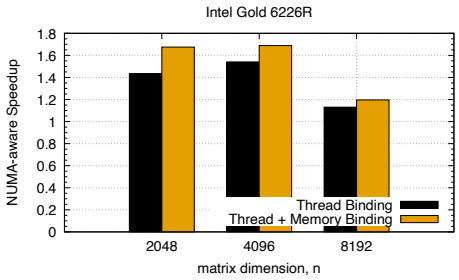

Figure 9: OpenMP pairwise speedup from NUMA optimizations with $n \in \{2048, 4096, 8192\}$ and $p = 32$.

ping provides access to the fast memory hierarchies on both CPUs. We observe speedups ranging from $1.05\times$ $(n = 4096, p = 2)$ to $1.33\times$ $(n = 2048, p = 16)$ when splitting threads (where $p \leq 16$) across sockets. We experimented with thread binding for the OpenMP triplet algorithm but not memory binding due to the irregular data dependencies. However, we did not observe significant performance improvements over the baseline, so we omit these results from Fig. 9. We obtain best OpenMP scaling when using the `untied` clause, which allows suspended tasks to be resumed on any available thread. Suspended tasks may cause additional reads from slow memory after restart. Hence, we do not expect NUMA optimizations to be helpful. We perform strong scaling experiments in Fig. 10 of the OpenMP variants under the same settings as for Fig. 9 and report self-relative efficiency achieved. We report efficiencies with and without NUMA optimizations. The pairwise algorithm without NUMA optimizations achieves efficiencies of $39.7\%, 43.8\%$, and 63.7% at $p = 32$ for $n = 2048, 4096$ and 8192, respectively. Including NUMA optimizations yields efficiencies of $57.5\%, 73.0\%$, and 82.0% for $p = 32$. The triplet algorithm achieves efficiencies of $36.6\%, 40.2\%$, and 54.3% without NUMA optimizations and $50.8\%, 46.8\%$, and 59.2% with NUMA optimizations for $p = 32$. The triplet algorithm is the faster

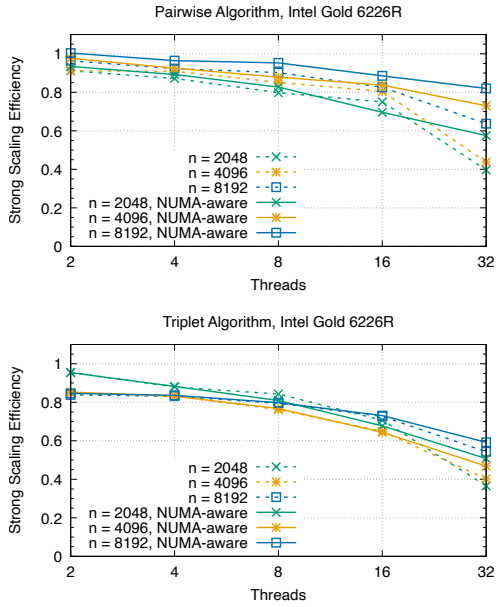

Figure 10: Self-relative strong scaling efficiency of OpenMP Pairwise (top) and Triplet (bottom).

sequential baseline, hence the OpenMP triplet efficiencies are lower than those reported for OpenMP pairwise. We also study weak scaling of the two algorithms with and without NUMA optimizations. We fix n^3/p over the range of p tested. We use the matrix sizes $n_1 \in \{2048, 4096, 8192\}$, where n_1 is the matrix size at $p = 1$. Figure 11 shows the results of the weak scaling experiments. The pairwise algorithm without NUMA optimizations attains weak scaling efficiencies of 78.5%, 78.8%, and 83.3% for $n_1 = 2048, 4096$, and 8192, respectively at 32 threads. With NUMA optimizations, the efficiencies increase to 78.5%, 87.9%, and 91.5% for each of the matrix size settings at $p = 32$. Triplet without NUMA optimizations achieves weak scaling efficiencies of $59.0\%, 69.0\%$, and 67.7% and $63.1\%, 68.9\%$, and 67.1% with NUMA optimizations at $p = 32$. For reference, MKL SGEMM attained 43.1% strong scaling efficiency at $p = 32$ for $n = 2048$ with OpenMP thread binding. The optimized sequential variants of PaLD attain up to 28% of peak hardware peak which is lower than sequential MKL SGEMM. PaLD strong scaling efficiencies are higher as a result.

7 Experiments with Application Data

7.1 Text Analysis We demonstrate the utility of PaLD on larger datasets than previously considered [2] for semantic analysis of words extracted from Shakespeare sonnets [12]. Words are converted to vectors using the pre-trained fastText word embedding [4, 13], yielding a dataset of 2712 words. We compute Eu-

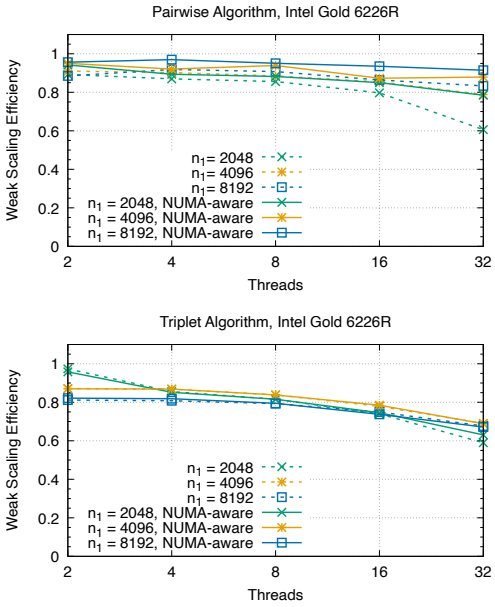

Figure 11: Self-relative weak scaling efficiency of OpenMP Pairwise (top) and Triplet (bottom).

Figure 12: Word clouds from PaLD analysis (left column) and distance analysis (right column) of the words *guilt* and *halt*. Font size is proportional to cohesion values and inverse distances.

clidean distance between embedding vectors and generate the cohesion matrix C using the OpenMP pairwise algorithm. Figure 12 shows words associated with *guilt* and *halt* obtained from PaLD and from analyzing only the distance matrix D. PaLD is parameter-free, with strong ties determined by a universal threshold (see [2]), whereas analysis using D requires a user-tuned distance or neighbor-count cutoff. Note the differing sizes of strong-tie neighborhoods between the two words. PaLD finds 20 words with strong ties to *guilt* and 5 words for *halt*. The 20 closest words to *guilt* based on distance correspond to a cutoff of 2.26. We observe significant overlap between the two sets, though PaLD reports stronger ties to *expiate* and *conscience*. PaLD finds 5 words with strong ties to *halt*. To illustrate the pitfalls of tuning an absolute distance threshold, we apply the distance cutoff 2.26 for *halt*, which yields 23 words including several unrelated ones (e.g. *just* and *say*). This suggests that absolute distance thresholds are not robust to varying density and distance scales within word neighborhoods. A distance cutoff of 2.14 is required for *halt* to match results obtained from PaLD. Applying the cutoff to *guilt* identifies only 8 related words, missing several words like *expiate*. We attain a speedup of 21× using the NUMA optimized OpenMP pairwise algorithm at $p = 32$ and an overall run time of 0.211 seconds.

Dataset	n	sequential	$p = 32$
ca-GrQc	5242	31.56	**1.186 (26.6×)**
ca-HepPh	12008	381.1	**13.35 (28.5×)**
ca-CondMat	23133	2827	**93.25 (30.3×)**

Table 2: Pairwise runtimes (in sec.) and maximum speedup over pairwise sequential on SNAP datasets.

7.2 Scaling on SNAP Datasets. We also perform scaling experiments on large datasets obtained from the SNAP data repository [17] to illustrate PaLD scalability on collaboration networks. We obtain distance matrices by computing all-pairs shortest path distances. Table 2 reports the running times (in seconds) and speedup achieved at $p = 32$ for the pairwise algorithm. We use the optimized sequential pairwise algorithm as our baseline. We achieve speedups of 26.6×, 28.5× and 30.3× on the ca-GrQC, ca-HepPh, and ca-CondMat datasets [16], respectively. For ca-CondMat, we are able to reduce the running time of computing C from 47 minutes (optimized pairwise sequential) to 93 seconds (OpenMP pairwise with $p = 32$).

8 Conclusion

This paper presents several sequential and shared-memory parallel algorithms for PaLD [2]. We prove that sequential variants are communication-optimal, up to constant factors. We illustrate that branch avoidance is critical to attaining high performance; achieving a speedup of up to 29× over naive sequential variants. Based on our theoretical and empirical studies, we conclude that the triplet variant is the faster sequential algorithm for large matrices due to less computation. However, we show that the pairwise algorithm is more amenable to parallelization due to regular data dependencies and load balance. We observe strong scaling speedups up to 26.2× (63.7% efficiency), and weak scaling efficiencies of up to 82.0% at $p = 32$. With the performance achieved on the text analysis and graph applications, we show that PaLD can be scaled to nearly any dataset with a distance matrix that fits in the memory of a single server.

Acknowledgements. We thank the anonymous reviewers for their feedback and helpful suggestions. We would like to thank Kenneth S. Berenhaut for helpful feedback on the presentation of PaLD and discussions on applying PaLD to semantic analysis of word embedding in Section 7. We would also like to thank Yixin Zhang for code contributions to preliminary versions of the pairwise algorithms. This work is supported by the National Science Foundation under Grant No. OAC-2106920 and the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research program under Award Number DE-SC-0023296.

References

[1] G. Ballard, E. Carson, J. Demmel, M. Hoemmen, N. Knight, and O. Schwartz, *Communication lower bounds and optimal algorithms for numerical linear algebra*, Acta Numerica, 23 (2014), p. 1–155, https://doi.org/10.1017/S0962492914000038.

[2] K. S. Berenhaut, K. E. Moore, and R. L. Melvin, *A social perspective on perceived distances reveals deep community structure*, Proceedings of the National Academy of Sciences, 119 (2022), p. e2003634119, https://doi.org/10.1073/pnas.2003634119.

[3] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel, *Optimizing matrix multiply using phipac: A portable, high-performance, ansi c coding methodology*, in ACM International Conference on Supercomputing, New York, NY, USA, 1997, Association for Computing Machinery, p. 253–260, https://doi.org/10.1145/2591635.2667174, https://doi.org/10.1145/2591635.2667174.

[4] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, *Enriching word vectors with subword information*, CoRR, abs/1607.04606 (2016), http://arxiv.org/abs/1607.04606, https://arxiv.org/abs/1607.04606.

[5] R. J. G. B. Campello, D. Moulavi, and J. Sander, *Density-based clustering based on hierarchical density estimates*, in Advances in Knowledge Discovery and Data Mining, J. Pei, V. S. Tseng, L. Cao, H. Motoda, and G. Xu, eds., Berlin, Heidelberg, 2013, Springer Berlin Heidelberg, pp. 160–172.

[6] R. J. G. B. Campello, D. Moulavi, A. Zimek, and J. Sander, *Hierarchical density estimates for data clustering, visualization, and outlier detection*, ACM Trans. Knowl. Discov. Data, 10 (2015), https://doi.org/10.1145/2733381, https://doi.org/10.1145/2733381.

[7] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, *A density-based algorithm for discovering clusters in large spatial databases with noise*, in Proceedings of the Second International Conference on Knowledge Discovery and Data Mining, KDD'96, AAAI Press, 1996, p. 226–231.

[8] J. Goldberger, G. E. Hinton, S. Roweis, and R. R. Salakhutdinov, *Neighbourhood components analysis*, in Advances in Neural Information Processing Systems, L. Saul, Y. Weiss, and L. Bottou, eds., vol. 17, MIT Press, 2004, https://proceedings.neurips.cc/paper_files/paper/2004/file/42fe880812925e520249e808937738d2-Paper.pdf.

[9] K. Goto and R. A. v. d. Geijn, *Anatomy of high-performance matrix multiplication*, ACM Trans. Math. Softw., 34 (2008), https://doi.org/10.1145/1356052.1356053.

[10] O. Green, J. Fox, A. Watkins, A. Tripathy, K. Gabert, E. Kim, X. An, K. Aatish, and D. A. Bader, *Logarithmic radix binning and vectorized triangle counting*, in 2018 IEEE High Performance extreme Computing Conference (HPEC), 2018, pp. 1–7, https://doi.org/10.1109/HPEC.2018.8547581.

[11] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Sixth Edition: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th ed., 2017.

[12] T. M. Inc., *Text analytics toolbox version: 1.9 (r2022b)*, 2022, https://www.mathworks.com.

[13] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov, *Bag of tricks for efficient text classification*, CoRR, abs/1607.01759 (2016), http://arxiv.org/abs/1607.01759, https://arxiv.org/abs/1607.01759.

[14] N. P. Jouppi, *Superscalar vs. superpipelined machines*, SIGARCH Comput. Archit. News, 16 (1988), pp. 71–80, https://doi.org/10.1145/48675.48686, https://doi.org/10.1145/48675.48686.

[15] N. P. Jouppi and D. W. Wall, *Available instruction-level parallelism for superscalar and superpipelined machines*, in ASPLOS-III Proceedings - Third International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, Massachusetts, USA, April 3-6, 1989, J. S. Emer and J. L. Hennessy, eds., ACM Press, 1989, pp. 272–282, https://doi.org/10.1145/70082.68207, https://doi.org/10.1145/70082.68207.

[16] J. Leskovec, J. Kleinberg, and C. Faloutsos, *Graph evolution: Densification and shrinking diameters*, ACM Trans. Knowl. Discov. Data, 1 (2007), p. 2–es, https://doi.org/10.1145/1217299.1217301, https://doi.org/10.1145/1217299.1217301.

[17] J. Leskovec and A. Krevl, *SNAP Datasets: Stanford large network dataset collection*. http://snap.stanford.edu/data, June 2014.

[18] OpenMP Architecture Review Board, *OpenMP application program interface version 5.2*, Nov. 2021, https://www.openmp.org/specifications/.

[19] T. M. Smith, B. Lowery, J. Langou, and R. A. van de Geijn, *A tight I/O lower bound for matrix multiplication*, Tech. Report 1702.02017, arXiv, 2019, https://arxiv.org/abs/1702.02017.