

# Sparse Tensor Transpositions

Suzanne Mueller  
MIT CSAIL  
suzmue@csail.mit.edu

Peter Ahrens  
MIT CSAIL  
pahrens@csail.mit.edu

Stephen Chou  
MIT CSAIL  
s3chou@csail.mit.edu

Fredrik Kjolstad  
Stanford University  
kjolstad@stanford.edu

Saman Amarasinghe  
MIT CSAIL  
saman@csail.mit.edu

## ABSTRACT

We present a new algorithm for transposing sparse tensors called QUESADILLA. The algorithm converts the sparse tensor data structure to a list of coordinates and sorts it with a fast multi-pass radix algorithm that exploits knowledge of the requested transposition and the tensors input partial coordinate ordering to provably minimize the number of parallel partial sorting passes. We evaluate both a serial and a parallel implementation of QUESADILLA on a set of 19 tensors from the FROSTT collection, a set of tensors taken from scientific and data analytic applications. We compare QUESADILLA and a generalization, TOP-2-SADILLA to several state of the art approaches, including the tensor transposition routine used in the SPLATT tensor factorization library. In serial tests, QUESADILLA was the best strategy for 60% of all tensor and transposition combinations and improved over SPLATT by at least 19% in half of the combinations. In parallel tests, at least one of QUESADILLA or TOP-2-SADILLA was the best strategy for 52% of all tensor and transposition combinations.

## CCS CONCEPTS

• **Mathematics of computing** → **Mathematical software performance**; • **Theory of computation** → **Sorting and searching**; • **Software and its engineering** → **Source code generation**.

## KEYWORDS

Sparse Tensors, Transposition, Sorting, COO, Radix Sort

## 1 INTRODUCTION

Tensors generalize vectors and matrices to any number of dimensions. Tensors used in computation are often sparse, which means many of the values are zero. To take advantage of the large number of zeroes in the tensor, we use sparse formats that allow the zeroes to be compressed away. These formats range from a simple list of coordinates to complicated data structures such as Compressed Sparse Row (CSR) [8], Doubly Compressed Sparse Row (DCSR) [4], Block Compressed Sparse Row (BCSR) [13], and Compressed Sparse Fiber (CSF) [23]. These formats have a natural ordering of their dimensions that provides a lexicographical ordering of the tensor nonzeros. In a sorted list of coordinates, the order of the sorting keys determines this lexicographic ordering.

Tensor algebra is used to compute with data stored in tensors. These multidimensional computations need to access the nonzero entries in one or more tensors, compute, and store the results. Accessing the nonzero entries requires some traversal of the tensor. However, unlike for dense tensors, traversing the nonzeros of a

sparse tensor in different lexicographical orderings may be asymptotically more expensive than the natural lexicographical ordering. Therefore, it is often faster to first transpose input tensors by re-ordering their dimensions before executing tensor expressions. This way, the tensor can be accessed naturally in the expression itself.

Tensor transposition is ubiquitous in data processing. Anytime multiple tensor expressions are composed and the output of one expression must be used as an input to the next, with a different index ordering and possibly a different sparse format, we need to transpose. For example, element-wise operations between tensors without matching index orderings (thus requiring transposition as a bottleneck) is listed as one of the five benchmark operations in the Parallel Sparse Tensor Algorithm Benchmark Suite (PASTA) [17]. Sparse tensor transposition may also occur when several different orderings of input data are required for efficient operation, but the space is not available to hold all of them. Such a situation might arise when using an alternating least squares method for canonical polyadic decomposition [23].

Prior work has focused extensively on dense tensor transpositions [6, 11, 14, 15, 20, 24–27]; we refer readers to [24] for a summary. Sparse matrix and tensor transposition have received relatively little attention [29]. A fast CSR sparse matrix transposition algorithm is proposed in [12], and improvements are proposed in [9, 29]. Further variations on sparse matrix transposition are discussed in [5, 10, 28, 30, 31]. None of these techniques, however, readily generalize to sparse tensor transposition with input tensors of arbitrary ranks.

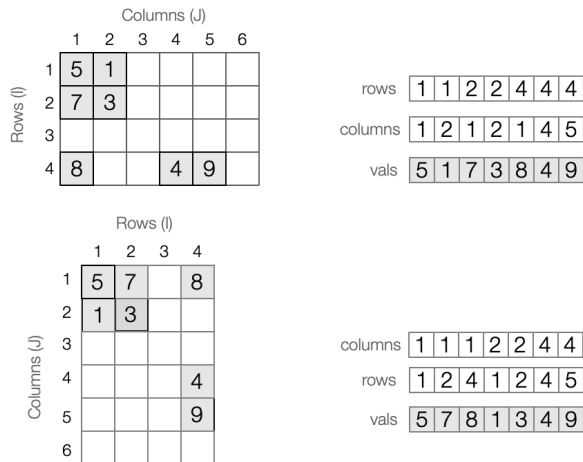
Tensors are often stored as a list of coordinates of nonzeros. If the coordinates are ordered lexicographically, adjacent coordinates may share the same indices in the first several modes. The Compressed Sparse Fiber (CSF) format [23] compresses these duplicate nonzeros using a tree-like storage format. In CSF, nodes represent indices, leaves represent nonzeros and paths from root to leaf represent coordinates. The children of each node are ordered. The matrix case of CSF is called Compressed Sparse Row (CSR).

More complicated sparse tensor formats like CSF often have similar ordering constraints, and require access to the coordinates in some lexicographic order in order to construct the tensor. The current state of the art for transposing sparse tensors involves converting the sparse tensor into a list of coordinates, sorting the list of coordinates, and finally packing the list of coordinates into the desired sparse tensor format [23].

This approach reduces the problem of transposing a tensor into a problem of sorting a list of coordinates. However, the lists of coordinates have partial orderings we can use to accelerate the sorting algorithms. Consider the example matrix in Figure 1. In

order to transpose the matrix, the column coordinates must be ordered lexicographically before the row coordinates. This could be accomplished by sorting with the column coordinate as the primary key and the row coordinate as the secondary key.

We can do better than that. The coordinates are already sorted on the row coordinates. By doing a stable sort on just the column coordinate, we get the same result. In this paper, we will generalize this optimization to arbitrary tensor transpositions.



**Figure 1: The matrix  $A$  can be represented as a list of coordinates including only the nonzero values. Transposing the tensor in this format switches the lexicographic ordering of the rows and columns, such that the columns appear first. The top list of coordinates represent the matrix in the top left. The bottom list of coordinates represent the transposed version of this matrix.**

The main contributions of this work are:

- (1) A decomposition of tensor transposition into parallelizable near-linear-work partial sorts (one of the two partial sorts is novel) that optionally respect previous partial orderings.
- (2) An algorithm that uses partial orderings in the original sparse tensor format to minimize the number of partial sorts required by the transposition algorithm. This relates the parallel span of radix sorting to partial orderings in the input.
- (3) A parallel implementation that demonstrates this transposition algorithm is competitive with, and often faster than, state of the art approaches.

## 2 BACKGROUND

A tensor of **rank**  $r$  is a multidimensional array that associates  $r$ -tuples (referred to as **coordinates**) with values, or **entries**. We refer to the  $k^{th}$  position in a coordinate as mode  $k$ . The size of a tensor is specified by an  $r$ -tuple of **dimensions**  $n$ , where each index  $i_k$  is an integer in the range  $1 \leq i_k \leq n_k$ .

Let  $N$  be the number of nonzero entries in our tensor. A tensor is **sparse** if most of its entries are zero. This has led to the development of sparse tensor storage formats that support efficient computation over only the nonzero entries. These formats range from a simple

sorted list of nonzero coordinates together with their values, the **Coordinates** (COO) format [2], to more complicated hierarchical mode-by-mode compression schemes such as **Compressed Sparse Row** (CSR) or **Compressed Sparse Fiber** (CSF) [8, 23]. All three of these formats induce a natural lexicographic ordering of the dimensions; iterating over the tensor in the natural order can be done very efficiently.

We define lexicographic ordering on  $r$ -tuples recursively using a tuple  $\sigma$  of modes in order of their priority. We consider the coordinate  $i = (i_1, i_2, \dots)$  to be less than the coordinate  $i' = (i'_1, i'_2, \dots)$  under the ordering  $\sigma$  in two cases. The first case is when  $i_{\sigma_1} < i'_{\sigma_1}$ . The second case is when both  $i_{\sigma_1} = i'_{\sigma_1}$  and  $i < i'$  under the ordering  $(\sigma_2, \sigma_3, \dots)$ . For completeness, we say that all tuples are considered equal under  $\sigma = ()$ . We will refer to the  $(1, 2, \dots, r)$  ordering of  $k$ -tuples as the **simple** ordering. We say an ordering is **complete** if it contains  $r$  distinct modes.

### 2.1 Coordinates (COO)

COO stores the nonzero coordinates in the tensor as a list of  $\sigma$ -sorted coordinates. Transposing a tensor in COO format is equivalent to reordering the coordinate list to a new complete ordering. This simplicity makes COO a popular format; it is the only sparse tensor format for the MATLAB Tensor Toolbox and TensorFlow libraries, and is used as an intermediate format during transpositions in the SPLATT library [1, 2, 23]. Since most sparse tensor formats can be converted to and from COO format and the format is readily sorted, we focus on transposing tensors in COO format.

The COO format can be implemented either with a list of lists (one for each mode) or as a list of coordinate tuples. We will use the latter for notational purposes. Thus, we store an array  $A$  in COO using two arrays,  $A.crd$  and  $A.val$ . The  $crd$  array is an array of coordinates, and  $val$  is an array of corresponding values. This requires  $O(r)$  bits to store each coordinate, so the total storage requirement for indices is  $O(r * N)$  bits. Figure 2 shows an example of COO storage.

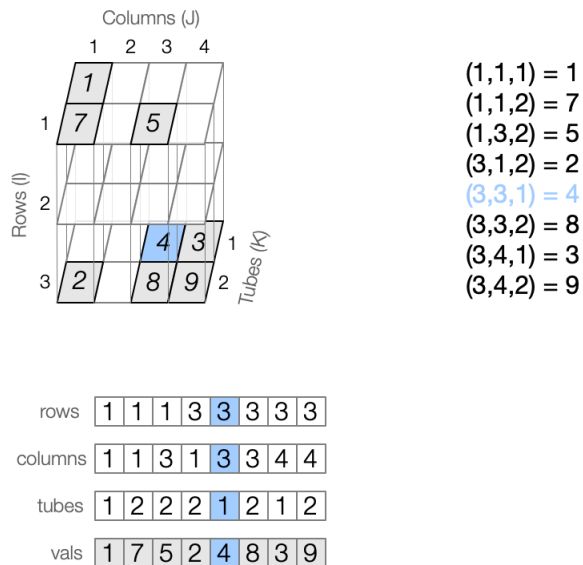
### 2.2 Transposition

The COO, CSR, and CSF formats are all sorted by lexicographic orderings on the coordinates. In COO, we sort the coordinates by some  $\sigma$  ordering, and in CSR and CSF,  $\sigma$  gives the order in which modes correspond to levels in the tree. CSR and CSF formats sort each level by the indices within that mode. In this work, **transposition** corresponds to a change in this storage order.

We will express tensor transposition operations using the final desired storage order  $\sigma$ . Without loss of generality, we assume the tensor is initially ordered by  $(1, 2, \dots)$ . For example, transposing a matrix stored in  $(1, 2)$  order is equivalent to changing the storage order to  $\sigma = (2, 1)$ , then relabeling the modes.

Certain computations will perform better when the dimensions can be iterated over efficiently in a different order than the initial storage order [16, 23]. When we encounter such a computation, it will be beneficial to transpose the tensor.

Tensors can be reordered using any sorting method. Since the coordinates come from a fixed range of values, we can also use sorts that work on fixed length keys, like histogram or radix sorts [7]. SPLATT, a sparse tensor library designed to be highly parallel,



**Figure 2: COO represents a tensor as a list of coordinates with all of the zero values compressed out. The list of arrays in the bottom left represent the tensor in the top left.**

uses a specialized sorting strategy to take advantage of potential parallelism that exists in the problem [23]. SPLATT chooses to first do a histogram sort on mode  $\sigma_1$ . It then sorts the coordinates for each index in mode  $i_{\sigma_1}$  using  $n_{\sigma_1}$  separate calls to quicksort. In the sequential implementation this strategy benefits from smaller subproblems for quicksort. In the parallel version, SPLATT is able to sort these buckets in parallel.

### 3 ALGORITHMS

A naive algorithm for sparse tensor transposition is to comparison sort the coordinates into the desired lexicographic order. However, since coordinates already have an initial ordering, we can think of sorting coordinates as simply changing the lexicographic ordering to prioritize different dimensions. It takes  $O(r)$  time to compare two coordinates of an  $r$ -tensor. Thus, a comparison based coordinate sort would run in  $O(rN \log N)$  time. However, since the indices are bounded by the dimensions, we can use parallelizable stable sorts like a histogram sort (called counting sort in [7, 19]) to sort the coordinates on a single mode  $k$  in  $O(rN + n_k)$  time. If we perform  $r$  histogram sorts (a radix sort on  $r$ -digit numbers), we can sort our coordinate list in  $O(r^2N + n_1 + n_2 + \dots)$  time, an asymptotic improvement over comparison sort when the dimensions are small. If we assume that coordinates are each processed in constant time, our histogram sort takes  $O(N + n_k)$  time and our radix sort takes  $O(rN + n_1 + n_2 + \dots)$  time. This algorithm can be improved further; for some transpositions, we do not need to perform all  $r$  sorts. For example, HALFPERM uses a histogram sort to prioritize the second dimension in the new ordering. Depending on the size of the second dimension, this single histogram sort is faster than a generic sort of the coordinates, and certainly faster than redundantly sorting the first dimension before sorting the second.

In this section, we formalize and generalize this idea to produce the Quesadilla tensor transposition algorithm, which provably performs the minimal number of histogram sorts. We start with a description of our histogram partial sort and a bucketing modification to produce two sorting primitives. We then use these primitives to build the Quesadilla and Top- $K$ -sadilla tensor transposition algorithms.

#### 3.1 Histogram Partial Sorts

A histogram sort sorts integer keys of bounded size. It first counts the number of occurrences of each key values. It then performs a prefix sum, also known as a cumulative sum or prefix scan, over the array of counts to determine where each group of equivalent keys will lie in the output array. This reserves enough space for all of the coordinates to appear in the output order, and the scanned array can be used record how full each output group is as the algorithm puts each coordinate directly into its output location. There is extensive research on the topic of parallelizing histogram sort as a subroutine of radix sort [19]. In our experiments, we use the same implementation as SPLATT [23], where each processor uses a private copy of *count* which is synchronized before moving coordinates to their output destinations.

---

#### Algorithm 1: PARTIALSORT( $A, ()$ , $\tau_k$ ) (Non-Bucketed)

---

**Input:**  $A$  is a rank- $r$  tensor of dimension  $n$  with  $N$  nonzeros stored in COO, sorted under the ordering  $\tau$ . Our goal is to sort  $A$  on  $p = \tau_k$ .

**Output:**  $A = B$ , a tensor in COO format sorted under the ordering

$$(\tau_k, \tau_1, \dots, \tau_{k-1}, \tau_{k+1}, \dots, \tau_r). \quad (1)$$

```

1  $count \leftarrow$  a length  $n_{\tau_k} + 1$  array of integers initialized to 0
2  $count[1] \leftarrow 1$ 
   // Compute the  $count$  array
3 for  $j \leftarrow 1$  to  $N$  do
4    $i = A.crd[j]$ 
5    $count[i_{\tau_k} + 1] \leftarrow count[i_{\tau_k} + 1] + 1$ 
6 end
   // Prefix sum
7 for  $i_{\tau_k} \leftarrow 2$  to  $n_{\tau_k} + 1$  do
8    $count[i_{\tau_k}] \leftarrow count[i_{\tau_k}] + count[i_{\tau_k} - 1]$ 
9 end
   // Move coordinates to final output destination
10 for  $j \leftarrow 1$  to  $N$  do
11    $i = A.crd[j]$ 
12    $j' = count[i_{\tau_k}]$ 
13    $B.crd[j'] = i$ 
14    $count[i_{\tau_k}] \leftarrow count[i_{\tau_k}] + 1$ 
15 end

```

---

The histogram sort iterates over the coordinates twice and the count array once. The total runtime is  $O(rN + n_{\tau_k})$ , where  $n_{\tau_k}$  is the dimension of the mode being sorted on. If we can process coordinates in constant time, the runtime is  $O(N + n_{\tau_k})$ . The histogram sort clearly produces a lexicographic ordering which prioritizes  $\tau_k$

first. Since the sort is stable, the relative ordering of other modes is unaffected. Thus, it moves the mode  $\tau_k$  to be the first mode in the lexicographic order, as described in (1).

### 3.2 Bucketed Histogram Sort

Although radix sort is most commonly performed from the least significant digit to the most significant digit, it will be useful for us to be able to work backwards sometimes, sorting one mode while respecting another partial ordering. Informally, we wish to sort a mode to a different position than the first spot in the output ordering. Formally, if our tensor is sorted with respect to  $\tau$ , we wish to sort on  $\tau_k$  while leaving the ordering  $(\tau_1, \dots, \tau_l)$  of the first  $l < k$  modes unaffected. This means that we need to sort each group of contiguous coordinates (a **bucket**) which agree on the values  $i_{\tau_1}, \dots, i_{\tau_l}$ . If we use a comparison-based sort within each bucket, we can perform the sort recursively but incur a logarithmic overhead. If we use a radix-based algorithm within each bucket, we need to perform an  $O(n_{\tau_k})$  prefix sum in each bucket. Since the number of buckets is bounded only by  $N$ , the resulting runtime of  $O(rNn_{\tau_k})$  is unacceptable.

Here, we describe a variation on histogram sort that discovers the buckets for  $(\tau_1, \dots, \tau_l)$ , sorts on  $\tau_k$ , then reimposes the previous ordering. Since there are at most  $N$  buckets, our algorithm runs in time  $O(N * (l + 1) + n_{\tau_k})$ . If we assume operations on coordinates occur in constant time, our algorithm runs in time  $O(N + n_{\tau_k})$ . Note that the input must be sorted under  $(\tau_1, \dots, \tau_l)$  to discover the buckets by examining adjacent coordinates.

Algorithms 1 and 2 are both called PARTIALSORT; we use Algorithm 1 when  $l = 0$  and Algorithm 2 otherwise.

Although we can save buckets as we fill the *count* array, Algorithm 2 performs an extra bucketing step to create the perm array, and the perm array introduces more indirection in the final bucketing step than the similar loop in Algorithm 1. Saving the buckets takes  $O(IN)$  time, the prefix sum takes  $O(n_{\tau_k})$  time, and the last two bucketing steps take  $O(rN)$  time. The total runtime of bucketed histogram sort is  $O(rN + n_{\tau_k})$ , or  $O(N + n_{\tau_k})$  if we assume constant-time operations on coordinates.

Bucketed histogram sort works by first stably sorting on mode  $\tau_k$ , then by sorting on  $(\tau_1, \dots, \tau_l)$  using the bucket array. After the loop on line 24, *perm* sorts  $A$  under (2)

$$(\tau_k, \tau_1, \dots, \tau_{k-1}, \tau_{k+1}, \dots, \tau_r).$$

Since the stored buckets correspond to equivalence classes of  $(\tau_1, \dots, \tau_l)$  in order, the loop on line 29 sorts  $A$  stably on the buckets, reprioritizing  $(\tau_1, \dots, \tau_l)$  in the ordering to produce the final order

$$(\tau_1, \dots, \tau_l, \tau_k, \tau_{l+1}, \dots, \tau_{k-1}, \tau_{k+1}, \dots, \tau_r).$$

As we describe parallelization strategies, we focus our attention on these three steps. Algorithm 2 discovers the buckets, stably sorts on the desired mode, then stably sorts on the buckets. Discovering the buckets is a simple linear-time algorithm that we can easily parallelize, taking care to account for buckets that cross processor boundaries. Most parallel implementations of histogram sort, including SPLATT, create private copies of the *count* array [23]. On  $P$  processors, these implementations run in  $O(N/P + n_{\tau_k})$  time. Since we can usually assume the dimension of the mode to be sorted is

---

#### Algorithm 2: PARTIALSORT( $A, (\tau_1, \dots, \tau_l), \tau_k$ ) (Bucketed)

---

**Input:**  $A$  is a rank- $r$  tensor of dimension  $n$  with  $N$  nonzeros stored in COO, sorted under the ordering  $\tau$ . Our goal is to sort  $A$  on  $\tau_k$  while maintaining the ordering  $(\tau_1, \dots, \tau_l)$ . We require that  $l < k$ .

**Output:**  $A = B$ , a tensor in COO format sorted under the ordering

$$(\tau_1, \dots, \tau_l, \tau_k, \tau_{l+1}, \dots, \tau_{k-1}, \tau_{k+1}, \dots, \tau_r). \quad (2)$$

```

1 count  $\leftarrow$  a length  $n_{\tau_k} + 1$  array of integers initialized to 0
2 count[1]  $\leftarrow$  1
3 bucket  $\leftarrow$  an uninitialized integer array of size  $N$ 
4 bucket[1]  $\leftarrow$  1
5 pos  $\leftarrow$  an uninitialized integer array of size  $N$ 
6 pos[1]  $\leftarrow$  1
7 perm  $\leftarrow$  an uninitialized integer array of size  $N$ 
8  $n' \leftarrow 1$ 
   // Compute the count array, equivalence classes
   // under  $(\tau_1, \dots, \tau_l)$ , and the pos array for each of
   // those classes
9  $i \leftarrow A.crd[1]$ 
10 count[ $i_{\tau_k} + 1$ ]  $\leftarrow$  count[ $i_{\tau_k} + 1$ ] + 1
11 for  $j \leftarrow 2$  to  $N$  do
12    $i \leftarrow A.crd[j]$ 
13    $i' \leftarrow A.crd[j - 1]$ 
14   if  $(i_{\tau_1}, \dots, i_{\tau_l}) \neq (i'_{\tau_1}, \dots, i'_{\tau_l})$  then
15      $n' \leftarrow n' + 1$ 
16     pos[ $n'$ ]  $\leftarrow j$ 
17   end
18   bucket[ $j$ ]  $\leftarrow n'$ 
19   count[ $i_{\tau_k} + 1$ ]  $\leftarrow$  count[ $i_{\tau_k} + 1$ ] + 1
20 end
   // Prefix sum
21 for  $i_{\tau_k} \leftarrow 2$  to  $n_{\tau_k} + 1$  do
22   count[ $i_{\tau_k}$ ]  $\leftarrow$  count[ $i_{\tau_k}$ ] + count[ $i_{\tau_k} - 1$ ]
23 end
   // Create permutation of  $A$  ordered on  $(\tau_k,)$ 
24 for  $j \leftarrow 1$  to  $N$  do
25    $i \leftarrow A.crd[j]$ 
26   perm[count[ $i_{\tau_k}$ ]]  $\leftarrow j$ 
27   count[ $i_{\tau_k}$ ]  $\leftarrow$  count[ $i_{\tau_k}$ ] + 1
28 end
   // Reintroduce the stored ordering on  $(\tau_1, \dots, \tau_l)$ 
29 for  $j \leftarrow 1$  to  $N$  do
30    $B.crd[pos[bucket[perm[j]]]] \leftarrow A.crd[perm[j]]$ 
31   pos[bucket[perm[ $j$ ]]]  $\leftarrow$  pos[bucket[perm[ $j$ ]]] + 1
32 end

```

---

small relative to the number of nonzeros, these parallel implementations of histogram sort are acceptable for sorting the desired mode. However, we cannot assume that the number of buckets is small relative to the number of nonzeros. To effectively parallelize the second sort, we would need to use an algorithm whose runtime is

linear in both the number of nonzeros and the range of keys to be sorted, such as a sample sort [3, 33]. Notice that the sampling step can be avoided because the bucket discovery step calculates the exact distribution of buckets (keys).

We can simplify parallelization of Algorithm 2 by decomposing the problem along bucket boundaries. The buckets limit the travel of coordinates between input and output orderings; coordinates do not escape their buckets. Therefore, running Algorithm 2 on a contiguous region of input buckets will compute the corresponding region of the output ordering. This gives our chosen parallel algorithm where we assign to each processor the buckets which begin in their region, and each processor simply runs Algorithm 2 locally on their section. Assuming that the buckets are small enough to permit effective decomposition, this algorithm also runs in time  $O(N/P + n_{\tau_k})$ . Notice that because SPLATT decomposes the local sorts along the index  $\sigma_1$ , SPLATT operates under the similar assumption that slices of the tensor are small enough to permit effective decomposition.

### 3.3 Bucketed Histogram Sort Example

We give an example of our bucketed histogram sort on a 4-tensor. For simplicity of presentation, we represent our coordinate list as 4-digit integers. The integers are initially sorted under the ordering (1, 2, 3, 4).

$$A.crd = [1218, 1224, 1274, 1421, 1437, 1456, 1472, 3216, 3283, 3286]$$

Suppose that we would like them to be sorted under the ordering  $\sigma = (1, 2, 4, 3)$ . We rearrange our digits to show the current ordering.

$$A.crd_{\sigma} = [1281, 1242, 1247, 1412, 1473, 1465, 1427, 3261, 3238, 3268]$$

Since our ordering doesn't change the first two digits, we can reorder  $A$  to be sorted under  $\sigma$  by bucketing on the first two digits. Our algorithm starts by discovering the buckets and computing the *bucket* and *pos* arrays, which store the numbers and positions of each bucket:

$$A.crd_{\sigma} = \underbrace{[1281, 1242, 1247]}_{\text{"12..."}} \underbrace{[1412, 1473, 1465, 1427]}_{\text{"14..."}} \underbrace{[3261, 3238, 3268]}_{\text{"32..."}}$$

$$bucket = [1, 1, 1, 2, 2, 2, 2, 3, 3, 3]$$

$$pos = [1, 4, 8, \text{undefined}, \dots]$$

Our counting sort sorts  $A$  by digit  $\sigma_3 = 4$ , producing:

$$perm = [4, 7, 9, 2, 3, 6, 8, 10, 5, 1]$$

$$A.crd[perm] =$$

$$[1421, 1472, 3283, 1224, 1274, 1456, 3216, 3286, 1437, 1218]$$

$$A.crd_{\sigma}[perm] =$$

$$[1412, 1427, 3238, 1242, 1247, 1465, 3261, 3268, 1473, 1281]$$

At this point, if we restrict our attention to one bucket at a time, the coordinates are sorted. We just need to put each element of  $A[perm]$  back into its corresponding bucket by sorting on  $bucket[perm]$ . The *pos* array functions as the *count* array does in counting sort.

$$B.crd = [1224, 1274, 1218, 1421, 1472, 1456, 1437, 3283, 3216, 3286]$$

$$B.crd_{\sigma} = \underbrace{[1242, 1247, 1281]}_{\text{"12..."}} \underbrace{[1412, 1427, 1465, 1473]}_{\text{"14..."}} \underbrace{[3238, 3261, 3268]}_{\text{"32..."}}$$

Notice that  $B.crd_{\sigma}$  is lexicographically ordered, as desired.

### 3.4 Minimizing Partial Sorts

Transposition via a full radix sort would consist of  $r$  calls to Algorithm 1. Not all transpositions, however, are equally difficult. For example, if we have a simply ordered 4-tensor and are asked to transpose it to the ordering (4, 1, 2, 3), this can be accomplished with the single call  $PARTIALSORT(A, (), 4)$ , as seen in (1). On the other hand, if we are asked to transpose to (4, 3, 2, 1), we show that this requires at least 3 calls  $PARTIALSORT$ , since the only relevant partial ordering we can use is that of the first mode. In this work, we generalize this insight to produce the QUESADILLA algorithm which transposes tensors to a given target ordering with the minimal number of calls to either Algorithm 1 or 2.

Although Algorithms 1 and 2 perform similar tasks, Algorithm 2 streams through and randomly accesses more vectors than Algorithm 1 does. If we count the number of unique vectors in each loop body separately (including initialization), Algorithm 1 streams through 4 vectors and randomly accesses 4 vectors, while Algorithm 2 streams through 7 vectors, and randomly accesses 7 vectors. While the costs of these algorithms are similar, they are not identical, and we should prefer to avoid the bucketed histogram variant whenever possible. For example, we can transpose to (2, 4, 1, 3) by calling  $PARTIALSORT(A, (), 2)$  and then  $PARTIALSORT(A, (2), 4)$ , but we can avoid a bucketed histogram sort by calling  $PARTIALSORT(A, (), 4)$  and then  $PARTIALSORT(A, (), 2)$ . Among transpositions that use the minimum number of partial sorts, we show that QUESADILLA uses the minimal number of bucketed partial sorts (Algorithm 2). Thus, our algorithm minimizes a cost model that weighs each pass equally, but breaks ties towards the non-bucketed variant.

We start by showing that for a given target order  $\sigma$ , we must sort on a certain set of modes and that in order to achieve the minimum number of sorts, some of these sorts must be bucketed. We then give an algorithm that only sorts on this necessary set of modes, and only uses bucketed sorts when required.

**3.4.1 Necessary Sorts.** The number of dimensions  $r$  is an upper bound on the number of passes needed to sort coordinates. This is the number of passes that are needed if we have a completely unsorted coordinate list and do a standard radix sort. The histogram sort and bucketed histogram sort can only move dimensions to the beginning of the lexicographic ordering. We use this fact to show a lower bound on the number of passes needed to sort the coordinates into the new lexicographic ordering. In several proofs, we will use a function  $f(\tau, p)$  that we define on complete  $r$ -orderings  $\tau$  as the set  $\{\tau_{k+1}, \dots, \tau_r\}$  where  $\tau_k = p$ . Thus,  $f(\tau, p)$  is the set of modes which follow  $p$  in the ordering  $\tau$ . For example,  $f((1, 3, 2, 4), 3) = \{2, 4\}$ .

**LEMMA 3.1.** *Let  $A$  be a list of  $r$ -coordinates ordered by the complete ordering  $\tau$ . Assume that  $A'$  is the  $\tau'$  ordered result of calling  $PARTIALSORT(A, (\tau_1, \dots, \tau_l), p)$  where  $p = \tau_k$  and  $k > l$ . If  $q \neq p$ , then  $f(\tau', q) \subseteq f(\tau, q)$ .*

**PROOF.** The result follows from a close examination of (1) and (2) which describe the output ordering of Algorithms 1 and 2. Let  $h$  be such that  $\tau_h = q$ . Note that  $k > l$ . If  $1 \leq h \leq l$  or  $k < h \leq r$ , then  $f(\tau', q) = f(\tau, q)$ . Otherwise,  $l < h < k$  and  $f(\tau', q) = f(\tau, q) \setminus p \subset f(\tau, q)$ .  $\square$

This idea that the set following some mode never expands when we sort on a different mode allows us to show that certain modes must be direct arguments to PARTIALSORT at some point in our sequence of calls that transposes the tensor.

LEMMA 3.2. *Let  $A$  be a list of  $r$ -coordinates ordered by the complete ordering  $\tau$ . Assume we wish to call PARTIALSORT some number of times to produce a  $\sigma$  ordering of  $A$ , and that  $f(\sigma, \sigma_i) \not\subseteq f(\tau, \sigma_i)$ . Consider any sequence of statements of the form*

$$A \leftarrow \text{PARTIALSORT}(A, (\psi_1, \dots, \psi_l), \psi_k),$$

where  $\psi$  is a complete intermediate ordering of  $A$ ,  $\psi_k \neq \sigma_i$ , and  $k > l$ . No such sequence will result in a  $\sigma$  ordering of  $A$ .

Therefore, any sequence of calls to SORT designed to return a  $\sigma$  ordering of  $A$  must include a call for each value of  $\sigma_i$  for which  $f(\sigma, \sigma_i) \not\subseteq f(\tau, \sigma_i)$ .

PROOF. At some point in our sequence of calls, assume that  $f(\sigma, \sigma_i) \not\subseteq f(\psi, \sigma_i)$ , and let  $\psi'$  be the ordering after the next call to PARTIALSORT. Lemma 3.1 implies that  $f(\psi', \sigma_i) \subseteq f(\psi, \sigma_i)$ , so it must still be the case that  $f(\sigma, \sigma_i) \not\subseteq f(\psi', \sigma_i)$ . Since we start with  $f(\sigma, \sigma_i) \not\subseteq f(\tau, \sigma_i)$ , there is no ordering in our sequence for which  $f(\sigma, \sigma_i) \subseteq f(\psi, \sigma_i)$ , and thus  $\psi$  can never equal  $\sigma$ .  $\square$

Lemma 3.2 implies a lower bound on the number of calls to PARTIALSORT required to transpose a  $\tau$ -ordered tensor  $A$  to  $\sigma$ -order. We refer to this number with the function  $b(\tau, \sigma)$ . We define  $b$  formally as the number of modes  $i$  for which  $f(\sigma, \sigma_i) \not\subseteq f(\tau, \sigma_i)$ . For example,  $b((1, 3, 2), (3, 1, 2)) = 1$ . While  $b$  gives us a lower bound on the number of sorts, it does not show a bound on whether each sort must be bucketed or not. We now show that no sequence of calls to PARTIALSORT of length  $b(\tau, \sigma)$  may include a call  $\text{PARTIALSORT}(A, (), p)$  if there exists  $i$  such that  $f(\sigma, \sigma_i) \subseteq f(\tau, \sigma_i)$  and  $p \in f(\sigma, \sigma_i)$ .

LEMMA 3.3. *Let  $A$  be a list of  $r$ -coordinates ordered by the complete ordering  $\tau$ . Consider any length  $b(\tau, \sigma)$  sequence of statements of the form*

$$A \leftarrow \text{PARTIALSORT}(A, (\psi_1, \dots, \psi_l), \psi_k),$$

where  $\psi$  is a complete intermediate ordering of  $A$  and  $k > l$ . If this sequence reaches the ordering  $\sigma$ , it may not contain any call where  $l = 0$  and there exists  $i$  such that  $f(\sigma, \sigma_i) \subseteq f(\tau, \sigma_i)$  and  $\psi_k \in f(\sigma, \sigma_i)$ .

PROOF. Lemma 3.2 implies that we must sort on each mode where  $f(\sigma, \sigma_k) \not\subseteq f(\tau, \sigma_k)$ . Since our sequence only involves  $b(\tau, \sigma)$  calls to PARTIALSORT, this sequence must only sort on these modes.

Assume for contradiction that our sequence involves a call where  $l = 0$  and there exists  $i$  such that  $f(\sigma, \sigma_i) \subseteq f(\tau, \sigma_i)$  and  $\psi_k \in f(\sigma, \sigma_i)$ . Let  $\psi'$  be the ordering following  $\psi$  after this call. Using (1), we see that  $f(\sigma, \sigma_i) \not\subseteq f(\psi', \sigma_i)$  since  $\psi_k \in f(\sigma, \sigma_i)$  but  $f(\psi', \sigma_i) = f(\psi, \sigma_i) \setminus \psi_k$ . Thus, Lemma 3.2 implies that we must sort on  $\sigma_i$  in order to achieve  $\sigma$  order, a contradiction as we are given that  $f(\sigma, \sigma_i) \subseteq f(\tau, \sigma_i)$ .  $\square$

Lemma 3.3 implies that if  $f(\sigma, \sigma_i) \subseteq f(\tau, \sigma_i)$ , a minimal sequence of sorts cannot involve non-bucketed sorts on modes in  $f(\sigma, \sigma_i)$ .

---

### Algorithm 3: QUESADILLASORT( $A, \sigma$ )

---

**Input:**  $A$  is any simply ordered list of  $r$ -coordinates,  $\sigma$  is an  $r$ -complete ordering.  
**Output:**  $A$ , sorted in  $\sigma$  order.

```

1  $l \leftarrow 0$ 
2 while  $l < r$  do
3    $k \leftarrow l$ 
4   while  $k + 1 < r$  and  $f(\sigma, \sigma_{k+1}) \not\subseteq f((1, 2, \dots), \sigma_{k+1})$  do
5      $k \leftarrow k + 1$ 
6    $l' \leftarrow k + 1$ 
7   while  $k > l$  do
8      $A \leftarrow \text{PARTIALSORT}(A, (\sigma_1, \dots, \sigma_l), \sigma_k)$ 
9      $k \leftarrow k - 1$ 
10   $l \leftarrow l'$ 
11 return  $A$ 
```

---

## 3.5 Quesadilla Sort

We now present the QUESADILLA algorithm for tensor transposition.

THEOREM 3.1. *Let  $A$  be a simply ordered list of  $r$ -coordinates. The sequence of sorts described by QUESADILLA( $\sigma$ ) will result in the  $\sigma$  ordered list of coordinates in  $A$ .*

PROOF. We prove the result by showing that before and after each execution of the body of the loop on line 7,  $A$  is sorted under a complete ordering  $\tau$ , where

$$(\tau_1, \dots, \tau_{l+l'-(k+1)}) = (\sigma_1, \dots, \sigma_l, \sigma_{k+1}, \dots, \sigma_{l'-1}), \quad (3)$$

and the remaining modes of  $\tau$  are in ascending order.

Before the first execution of our loop body,  $A$  is simply ordered,  $l = 0$ , and  $l' = k + 1$ . Thus, our claim is initially satisfied.

Assume our claim holds before some execution of the loop body. Let  $A'$  and  $k'$  be the values of  $A$  and  $k$  after executing the loop body. Let  $t$  be the mode such that  $\sigma_k = \tau_t$ . Since  $k > l$ , (3) implies that  $t > l + l' - (k + 1)$ . Combining this observation with (1) and (2) leads to the observation that  $A'$  is sorted under the complete ordering  $\tau'$ , where

$$\tau' = (\sigma_1, \dots, \sigma_l, \sigma_k, \dots, \sigma_{l'-1}, \tau_{l+l'-(k+1)}, \dots, \tau_{t-1}, \tau_{t+1}, \dots, \tau_r).$$

Therefore, (3) still holds for  $\tau'$  and  $k'$ . Because  $(\tau_{l+l'-k}, \dots, \tau_r)$  was ascending,  $(\tau'_{l+l'-k'}, \dots, \tau'_r)$  is also ascending. Thus, the claim holds after the execution of the loop body on line 7.

All that remains to be shown is that our claim holds after we move through the loop on line 2. After leaving the line 7 loop,  $k = l$  and  $A$  is sorted under the complete ordering

$$\tau = (\sigma_1, \dots, \sigma_{l'-1}, \tau_{l'}, \dots, \tau_r).$$

Notice that now  $f(\sigma, \sigma_{k+1}) \subseteq f((1, 2, \dots), \sigma_{k+1})$ , either because that was the condition that stopped the loop on line 4 or because that loop stopped when  $l' = r$  and  $f(\sigma, \sigma_r) = \emptyset$ . Thus, for all  $j > l'$ ,  $\sigma_j > \sigma_{l'}$ , and since  $(\tau_{l'}, \dots, \tau_r)$  is ascending,  $\tau_{l'} = \sigma_{l'}$ . Thus, we set  $l$  to  $l'$  and we have  $(\tau_1, \dots, \tau_l) = (\sigma_1, \dots, \sigma_l)$  when we reach line 7. The other claims will hold because  $k + 1$  will be equal to  $l'$ .  $\square$

**THEOREM 3.2.** *Given a target ordering  $\sigma$ , QUESADILLASORT( $\sigma$ ) uses the minimum-length sequence of calls to PARTIALSORT required to sort any simply ordered list of  $r$ -coordinates to  $\sigma$  order.*

**PROOF.** QUESADILLASORT only calls PARTIALSORT on modes  $\sigma_k$  where  $f(\sigma, \sigma_k) \not\subseteq f((1, 2, \dots), \sigma_k)$ . Thus, Lemma 3.2 implies that QUESADILLASORT makes the minimum number of required calls to PARTIALSORT.  $\square$

**THEOREM 3.3.** *Among minimum-length sequences of PARTIALSORT calls that sort simply ordered lists of  $r$ -coordinates to target ordering  $\sigma$ , the sequence used by QUESADILLASORT( $\sigma$ ) minimizes the number of bucketed partial sorts.*

**PROOF.** The first execution of the loop on line 4 stops once  $f(\sigma, \sigma_{k+1}) \subseteq f((1, 2, \dots), \sigma_{k+1})$ . For all  $j > k + 1$ , we have  $\sigma_j \in f(\sigma, \sigma_{k+1})$ . Since QUESADILLA uses non-bucketed sorts for all sorts on modes  $\sigma_1$  through  $\sigma_{k+1}$ , Lemma 3.3 implies that QUESADILLA uses the minimum number of bucketed partial sorts possible when the total number of sorts is minimized.  $\square$

### 3.6 Top- $K$ -sadilla Sort

Although the two sorting primitives presented are both histogram sort variants, they could be replaced with any stable sort such as quicksort or merge sort. However, if a comparison sort is used at some level  $k$  where the current ordering is  $\tau$  and  $(\tau_1, \dots, \tau_l) = (\sigma_1, \dots, \sigma_l)$ , it makes more sense to completely sort each bucket (equivalence class under  $(\tau_1, \dots, \tau_l)$ ) to  $\sigma$  order.

Thus, we propose the TOP- $K$ -SADILLA algorithm, which uses QUESADILLA to sort the tensor to  $(\sigma_1, \dots, \sigma_K)$  order, then sorts each bucket using quicksort. The best choice of the value  $K$  will be investigated in our experiments, since it depends both on the permutation and on the dimension of the tensor.

## 4 EVALUATION

We evaluate QUESADILLA and TOP- $K$ -SADILLA sort against various state of the art approaches for sparse tensor transposition. As we will show, on the whole, our technique outperforms these existing approaches.

### 4.1 Experimental Setup

We created both parallel and serial implementations of our technique. We implemented the serial version in a code generator that emits C++ code to transpose sparse tensors stored in the COO format using either QUESADILLA or TOP- $K$ -SADILLA sort. We implemented the parallel version by implementing parallel counting sort and bucketed counting sort primitives and calling the necessary sorts for QUESADILLA. To implement the quicksort portion of TOP- $K$ -SADILLA, we identified the buckets in parallel and then sorted each bucket using the OpenMP for-loop parallelization construct. The buckets were scheduled using dynamic scheduling for TOP-1-SADILLA and guided scheduling for TOP- $K$ -SADILLA when  $K > 1$ . We made these scheduling choices because we expected more smaller buckets when buckets correspond to more coordinates. The overhead for dynamically scheduling many small buckets caused significant slow down.

**Table 1: Statistics about tensors used in our experiments.**

Tensor	Nonzeros	Dimensions
flickr-3d	112890310	$319686 \times 28153045 \times 1607191$
nell-1	143599552	$2902330 \times 2143368 \times 25495389$
nell-2	76879419	$12092 \times 9184 \times 28818$
vast-2015-mc1-3d	26021854	$165427 \times 11374 \times 2$
chicago-crime-comm	5330673	$6186 \times 24 \times 77 \times 32$
delicious-4d	140126220	$532924 \times 17262471 \times 2480308 \times 1443$
enron	54202099	$6066 \times 5699 \times 44268 \times 1176$
flickr-4d	112890310	$319686 \times 28153045 \times 1607191 \times 731$
nips	3101609	$2482 \times 2862 \times 14036 \times 17$
uber	3309490	$183 \times 24 \times 1140 \times 1717$
lbnl-network	1698825	$1605 \times 4198 \times 1631 \times 4209 \times 868131$
vast-2015-mc1-5d	26021945	$165427 \times 11374 \times 2 \times 100 \times 89$

Our serial implementation is available at <https://github.com/suzmue/taco/tree/transpose> and our parallel implementation is available at <https://github.com/suzmue/splatt>.

To evaluate our technique, we compare it against SPLATT [23], a high-performance C++ toolkit for sparse tensor factorization that uses a combination of histogram sort, quicksort, and insertion sort to sort tensors in COO. We also evaluate against sparse tensor transposition routines that sort nonzeros with (least significant digit) radix sort (using Algorithm 1 for each pass) or glibc’s implementation of qsort.

We ran all experiments on a 2.5 GHz Intel Xeon E5-2680 v3 machine with 24 cores, 30 MB of L3 cache and 128 GB of main memory. The machine runs Ubuntu 18.04.3 LTS with glibc 2.27. We compiled the benchmarks using GCC 7.4.0. We ran each experiment 100 times and report minimum execution times.

We ran our experiments on real-world tensors obtained from the FROSTT Tensor Collection [22]. Table 1 reports statistics about these tensors. We stored tensors in the COO format and stored coordinates of nonzeros using 32-bit integers.

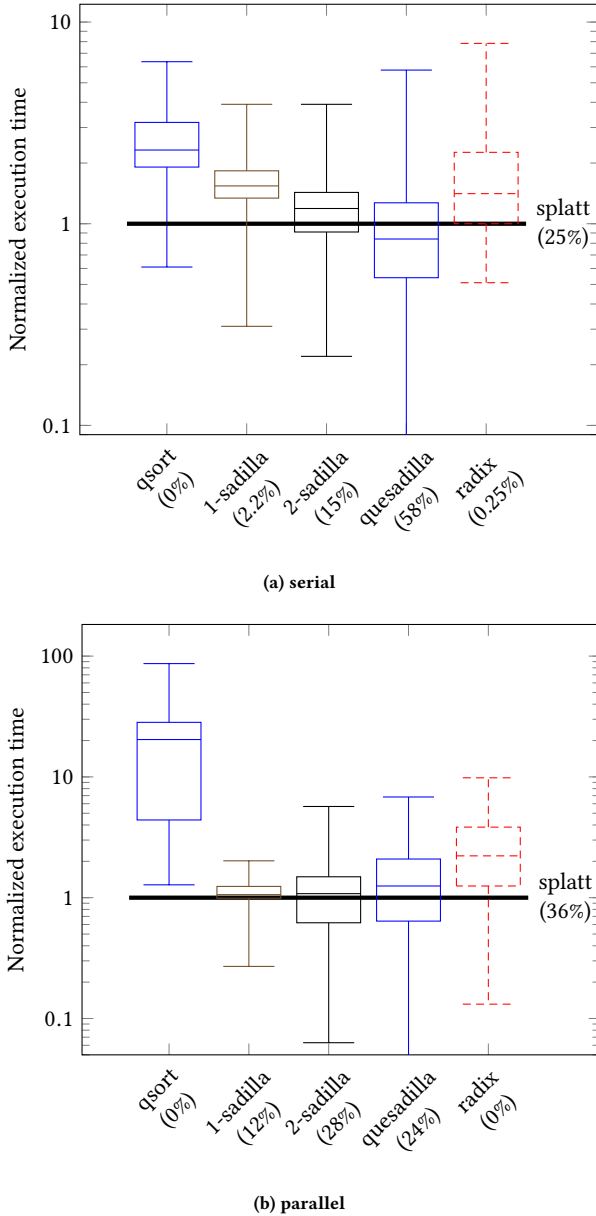
### 4.2 Performance Evaluation

For each tensor in Table 1, we measured the normalized running times of SPLATT, qsort, TOP- $K$ -SADILLA, QUESADILLA, and radix sort for transposing the tensor from its initial ordering  $\sigma = (1, \dots, r)$  to every  $r!$  possible ordering. Figure 3 shows the results of these experiments aggregated over all 408 possible combinations of input tensors and output orderings. The appendix includes more detailed results that show the performance of each algorithm for every combination of input tensor and output ordering.

In serial tests, these results demonstrate that QUESADILLA outperforms SPLATT, radix sort, and qsort on 60% of the sparse tensor transpositions. For half of all combinations, QUESADILLA is at least 1.19 $\times$  faster than SPLATT, 1.68 $\times$  faster than radix sort, and 2.76 $\times$  faster than qsort. In parallel tests, at least one of QUESADILLA or TOP-2-SADILLA was the best strategy for 52% of all tensor and transposition combinations.

QUESADILLA is able to significantly outperform radix sort by minimizing the number of passes over the input tensor. As Figure 4 shows, QUESADILLA exploits the partial ordering of the input tensor





**Figure 3: Normalized execution times of sparse tensor transposition with various algorithms, aggregated over all 408 possible combinations of test tensors and output orderings. Percentages in parentheses indicate the proportion of combinations for which each algorithm is the fastest. Results are normalized to SPLATT (horizontal line) for each tensor and output ordering. TOP-1-SADILLA denotes TOP-K-SADILLA with  $K = 1$ .**

to eliminate at least one sorting pass for all possible output orderings and eliminate two or more passes for the majority of output orderings. By contrast, radix sort always makes as many sorting passes as there are modes in the input tensor, thereby incurring overhead from unnecessary memory traffic.

QUESADILLA’s performance, however, depends to a large degree on the dimensions of the input tensor as well as the ordering of modes in the output. In particular, QUESADILLA is more efficient when it does not have to sort large modes. Figure 5, for instance, shows QUESADILLA’s performance for the lbnl-network tensor, whose last mode is significantly larger than the other modes. For output orderings where QUESADILLA does not have to sort the last mode, QUESADILLA significantly outperforms all other algorithms we evaluate. On the other hand, SPLATT and TOP-K-SADILLA (where  $K < r$ ) are more efficient for the other output orderings, with both being faster than QUESADILLA in approximately two-thirds of cases where QUESADILLA must sort the last mode in the serial implementation, and nearly all cases in the parallel implementation. This can be attributed to the fact that each invocation of PARTIALSORT in QUESADILLA requires a histogram containing  $n_k$  bins, where  $n_k$  is the size of the mode being sorted. When  $n_k$  is large, accesses into the histogram are less likely to hit the cache, thereby limiting performance. Furthermore, constructing the histogram incurs  $O(n_k)$  overhead, which becomes more significant when  $n_k$  is large. Thus, as Figure 6 shows, PARTIALSORT is significantly slower for large modes than for small modes, assuming the bucketed dimensions are the same. This, in turn, limits QUESADILLA’s performance for input tensors and output orderings that require sorting large modes. By contrast, SPLATT and TOP-K-SADILLA use comparison-based sorting algorithms to sort all but the first mode or the first several modes respectively, thus making their performance less dependent on the dimensions of the input tensor.

When  $K = 1$ , TOP-K-SADILLA reduces to the TOP-1-SADILLA algorithm that is similar to what SPLATT implements for sorting COO tensors, which we summarize in Section 2.2. Unlike SPLATT, which uses a custom hand-optimized implementation of quicksort, serial TOP-1-SADILLA uses qsort from C stdlib to sort nonzeros within each bucket created by the initial histogram sort. As Figure 3 shows, serial SPLATT outperforms serial TOP-1-SADILLA for most tensor transpositions in our experiments, thereby demonstrating that SPLATT’s custom implementation of quicksort is more efficient than qsort. This performance difference suggests we can improve TOP-K-SADILLA’s performance by using more optimized implementations of comparison sort to sort each bucket.

## 5 CONCLUSION

We have described an algorithm to transpose sparse tensors faster than simply sorting a list of coordinates. By taking advantage of the lexicographic ordering of the input and knowledge of the requested transposition, our algorithm applies only a subset of the passes of a radix sort and thereby reduces the amount of work required to sort the coordinates. We provide two non-comparison based partial sorting algorithms for radix sort passes that are optimized for different situations. We prove two things: (1) We prove that our algorithm minimizes the total calls to either sorting algorithm. (2) We prove that among sorts with the minimum total calls, we



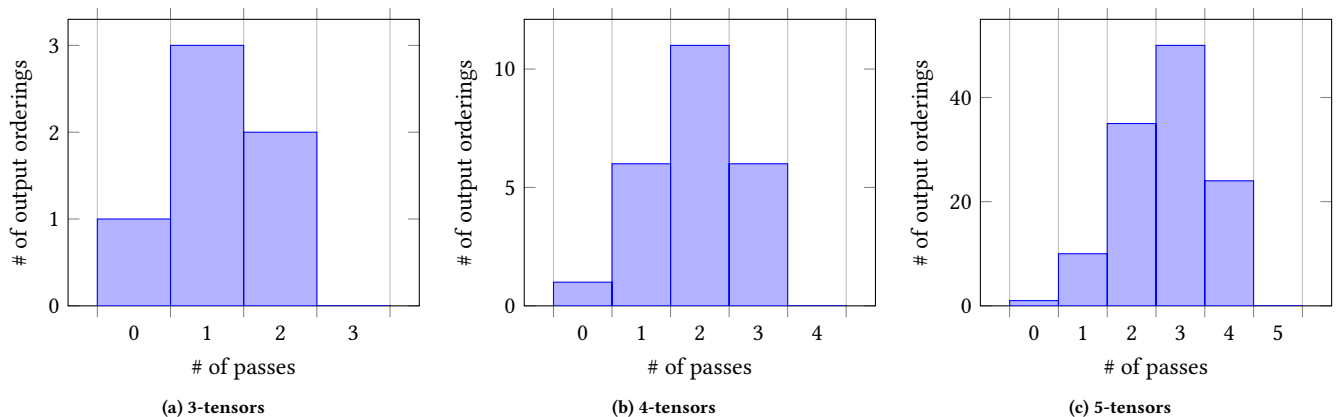


Figure 4: Distributions of the number of sorting passes needed by QUESADILLA to transpose tensors of varying rank.

minimize the number of calls to the more expensive of the two. The amount of work required by our algorithm is proportional to the number of modes that need reordering in the transposition. We evaluated our algorithm empirically with a C++ implementation, and showed that it produced significant improvements over existing approaches.

As sparse tensor representations receive increasing study, diversity in tensor formats will increase and applications will more frequently convert between formats. Sparse tensor transposition is the most basic instance of sparse format conversion, and an important subroutine in several format conversions. We have provided evidence that naive algorithms for sparse tensor transpositions can be improved substantially, but there are further improvements that need investigation.

Focusing on the multi-pass coordinate-sorting-based transposition technique we describe, improvements can be made in scheduling passes, the implementation of passes themselves, and handling the buckets. Although we minimize the number of passes over the data, we don't necessarily pick a schedule of passes that minimizes the true runtime. Since the bucketed histogram sort costs more than the histogram sort, we can improve our scheduling by minimizing a cost model which reflects the true costs of the passes.

We can improve the implementation of a sorting pass by reducing the size of coordinates using bit-packing techniques. If the mode to be sorted has a large dimension, it can make sense to perform the histogram sort itself as a radix sort, with multiple passes and a radix smaller than the dimension. In some cases, we can also fuse the first loop of the next histogram sort into the last loop of the current one, reducing the number of reads.

Discovering the buckets is expensive. If we need to perform several bucketed histogram sorts with the same buckets, we only need to discover the buckets once at the beginning, perform the histogram sorts, and then sort on the buckets at the end, skipping the bucketing step between the two sorts. Since  $l$  is constant, we can use the same buckets for all iterations of the loop on line 7 of Algorithm 3. Additionally, instead of evaluating the all  $l$  entries of each coordinate to discover the buckets, we can use buckets from the previous pass, which differ precisely when the previous entries

differed. While such an optimization would involve permuting a bucket array, we can avoid examining entire coordinates during bucket discovery, saving a factor of  $r$  in the asymptotic analysis.

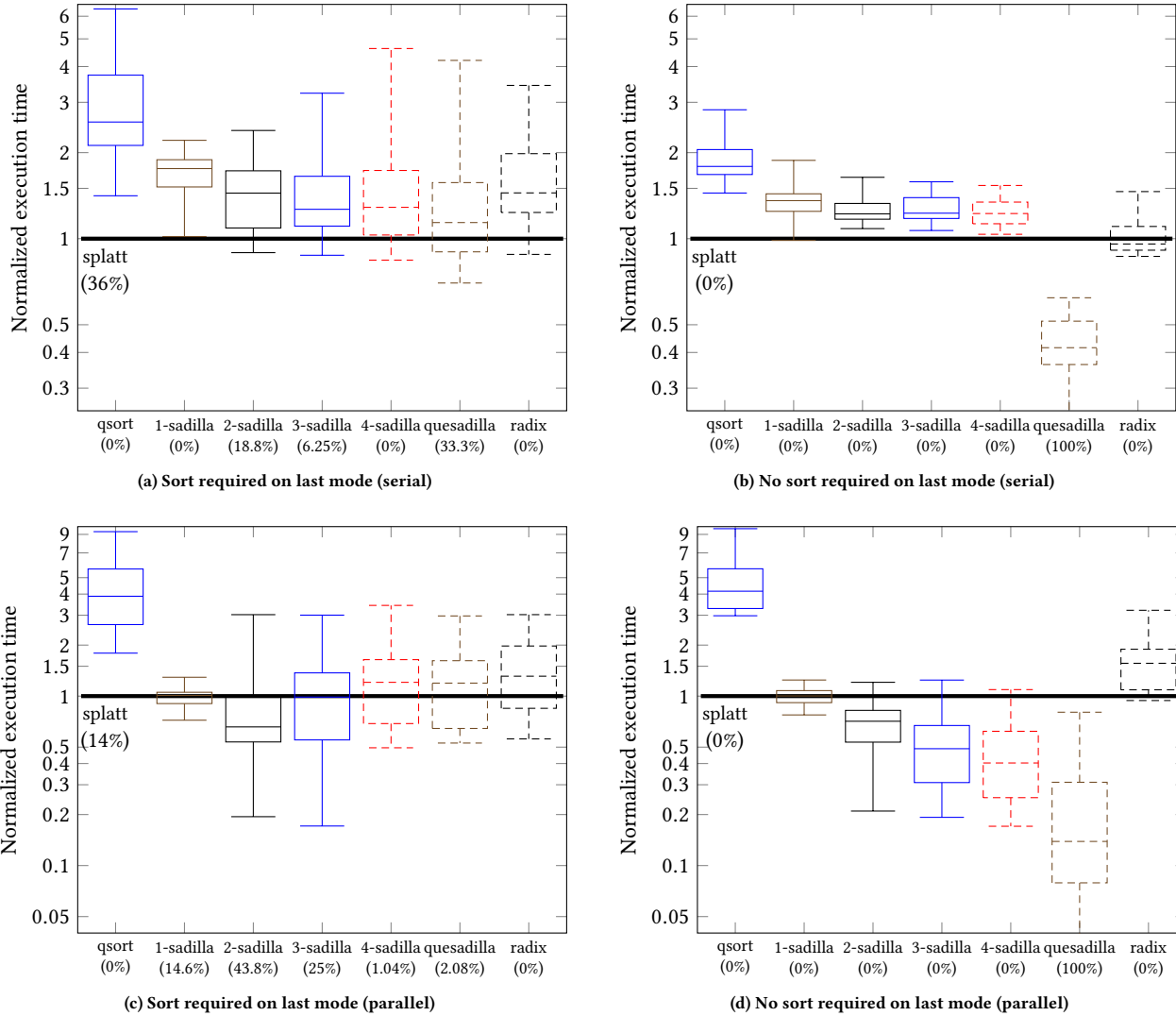
If our goal is to transpose tensors stored in formats other than COO, including formats like HiCOO [18], BICRS [32], and JAD [21], additional optimizations may present themselves. For example, instead of converting to coordinates, then sorting, our first histogram sort can iterate over the input format in order, fusing the conversion to coordinates into the first histogram sort. Additionally, the sorting techniques we describe in this work may apply directly to the format we want to transpose. If the tensor is in CSF, for example, it may be possible to sort the nodes in the CSF tree directly, moving the nodes instead of moving entire subtrees.

## ACKNOWLEDGMENTS

This work was supported by a grant from the Toyota Research Institute, DARPA PAPP Grant HR00112090017, and a DOE CSGF Fellowship DE-FG02-97ER25308.

## REFERENCES

- [1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *arXiv:1603.04467 [cs]* (March 2016). <http://arxiv.org/abs/1603.04467> arXiv: 1603.04467.
- [2] Brett W. Bader and Tamara G. Kolda. 2007. Efficient MATLAB Computations with Sparse and Factored Tensors. *SIAM Journal on Scientific Computing* 30, 1 (Jan. 2007), 205–231. <https://doi.org/10.1137/060676489>
- [3] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. 1991. A comparison of sorting algorithms for the connection machine CM-2. In *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures (SPAA '91)*. Association for Computing Machinery, Hilton Head, South Carolina, USA, 3–16. <https://doi.org/10.1145/113379.113380>
- [4] Aydin Buluc and John R. Gilbert. 2008. On the representation and multiplication of hypersparse matrices. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. 1–11. <https://doi.org/10.1109/IPDPS.2008.4536313> ISSN: 1530-2075.



**Figure 5: Normalized execution times of sparse tensor transposition with various algorithms for the lbnl-network tensor, aggregated over (a) all output orderings where QUESADILLA needs to sort on the last mode and (b) all output orderings where the last mode does not need to be sorted. Percentages in parentheses indicate the proportion of combinations for which each algorithm is the fastest. Results are normalized to SPLATT (horizontal lines) for each tensor and output ordering. Again, e.g. TOP-1-SADILLA denotes TOP-K-SADILLA with  $K = 1$ .**

[5] Frank Cameron. 1993. Two space-saving algorithms for computing the permuted transpose of a sparse matrix. *Advances in Engineering Software* 17, 1 (Jan. 1993), 49–60. [https://doi.org/10.1016/0965-9978\(93\)90041-Q](https://doi.org/10.1016/0965-9978(93)90041-Q)

[6] Bryan Catanzaro, Alexander Keller, and Michael Garland. 2014. A decomposition for in-place matrix transposition. *ACM SIGPLAN Notices* 49, 8 (Feb. 2014), 193–206. <https://doi.org/10.1145/2692916.2555253>

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to algorithms* (3rd ed ed.). MIT Press, Cambridge, Mass. OCLC: ocn311310321.

[8] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman. 1982. Yale sparse matrix package I: The symmetric codes. *Internat. J. Numer. Methods Engrg.* 18, 8 (1982), 1145–1151. <https://doi.org/10.1002/nme.1620180804>

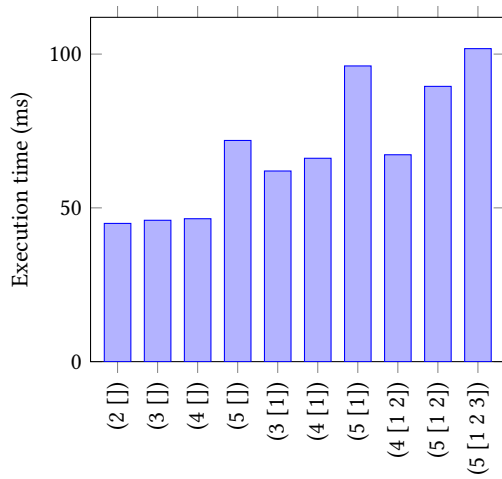
[9] Miguel A. Gonzalez-Mesa, Eladio D. Gutierrez, and Oscar Plata. 2013. Parallelizing the Sparse Matrix Transposition: Reducing the Programmer Effort Using Transactional Memory. *Procedia Computer Science* 18 (Jan. 2013), 501–510. <https://doi.org/10.1016/j.procs.2013.05.214>

[10] Song Guo, Yong Dou, Yuanwu Lei, Qiang Wang, Fei Xia, and Jianing Chen. 2016. Designing Parallel Sparse Matrix Transposition Algorithm Using ELLPACK-R for GPUs. In *Computer Engineering and Technology (Communications in Computer and Information Science)*, Weixia Xu, Liqun Xiao, Jinwen Li, and Chengyi Zhang (Eds.), Springer, Berlin, Heidelberg, 61–68. [https://doi.org/10.1007/978-3-662-49283-3\\_7](https://doi.org/10.1007/978-3-662-49283-3_7)

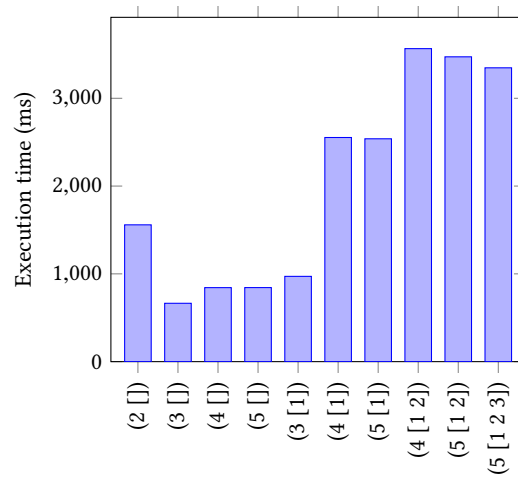
[11] Fred Gustavson, Lars Karlsson, and Bo Kågström. 2012. Parallel and Cache-Efficient In-Place Matrix Storage Format Conversion. *ACM Transactions on Mathematical Software (TOMS)* 38, 3 (April 2012), 17:1–17:32. <https://doi.org/10.1145/2168773.2168775>

[12] Fred G. Gustavson. 1978. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Transactions on Mathematical Software (TOMS)* 4, 3 (Sept. 1978), 250–269. <https://doi.org/10.1145/355791.355796>

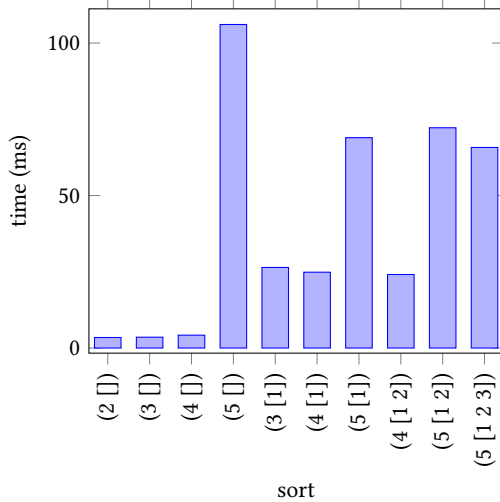
[13] Eun-Jin Im and Katherine Yelick. 2001. Optimizing Sparse Matrix Computations for Register Reuse in SPARSITY. In *Computational Science – ICCS 2001*, Vassil N. Alexandrov, Jack J. Dongarra, Benjoe A. Juliano, René S. Renner, and C. J. Kenneth



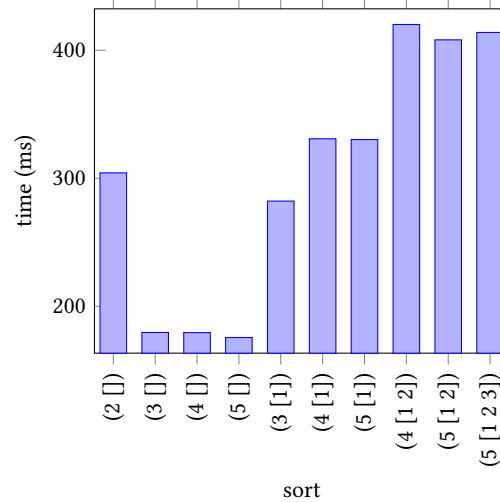
(a) lbnl-network (serial)



(b) vast-2015-mc1-5d (serial)



(c) lbnl-network (parallel)



(d) vast-2015-mc1-5d (parallel)

**Figure 6: Execution times of PARTIALSORT for sorting different modes of two test tensors. Labels along the x-axes indicate the modes being sorted; for instance, (4 [1 2]) denotes sort on mode 4 assuming modes 1 and 2 are bucketed.**

- Tan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 127–136.
- [14] Lars Karlsson. 2009. *Blocked in-place transposition with application to storage format conversion*. Technical Report.
- [15] S. D. Kaushik, C. Huang, J. R. Johnson, R. W. Johnson, and P. Sadayappan. 1993. Efficient transposition algorithms for large matrices. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*. 656–665. <https://doi.org/10.1109/SUPERC.1993.1263520>
- [16] Fredrik Kjolstad, Peter Ahrens, Shoab Kamil, and Saman Amarasinghe. 2018. Sparse Tensor Algebra Optimizations with Workspaces. *arXiv:1802.10574 [cs]* (April 2018). <http://arxiv.org/abs/1802.10574> arXiv: 1802.10574.
- [17] Jiajia Li, Yuchen Ma, Xiaolong Wu, Ang Li, and Kevin Barker. 2019. PASTA: a parallel sparse tensor algorithm benchmark suite. *CCF Transactions on High Performance Computing* 1, 2 (Aug. 2019), 111–130. <https://doi.org/10.1007/s42514-019-00012-w>
- [18] Jiajia Li, Jimeng Sun, and Richard Vuduc. 2018. HiCOO: Hierarchical Storage of Sparse Tensors. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 238–252. <https://doi.org/10.1109/SC.2018.00022> ISSN: null.
- [19] Omar Obeya, Endrias Kahssay, Edward Fan, and Julian Shun. 2019. Theoretically-Efficient and Practical Parallel In-Place Radix Sorting. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '19)*. Association for Computing Machinery, Phoenix, AZ, USA, 213–224. <https://doi.org/10.1145/3323165.3323198>
- [20] Greg Ruetsch and Paulius Micikevicius. 2009. *Optimizing matrix transpose in CUDA*. Technical Report.
- [21] Youcef Saad. 1989. Krylov Subspace Methods on Supercomputers. *SIAM J. Sci. Statist. Comput.* 10, 6 (Nov. 1989), 1200–1232. <https://doi.org/10.1137/0910073>
- [22] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. 2017. FROSTT: The Formidable Repository of Open Sparse Tensors and Tools. <http://frostt.io/>
- [23] Shaden Smith, Niranjan Ravindran, Nicholas D. Sidiropoulos, and George Karypis. 2015. SPLATT: Efficient and Parallel Sparse Tensor-Matrix Multiplication. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS '15)*. IEEE Computer Society, Washington, DC, USA, 61–70. <https://doi.org/10.1109/IPDPS.2015.27>

- [24] Paul Springer, Jeff R. Hammond, and Paolo Bientinesi. 2017. TTC: A High-Performance Compiler for Tensor Transpositions. *ACM Transactions on Mathematical Software (TOMS)* 44, 2 (Aug. 2017), 15:1–15:21. <https://doi.org/10.1145/3104988>
- [25] I. Sung, G. D. Liu, and W. W. Hwu. 2012. DL: A data layout transformation system for heterogeneous computing. In *2012 Innovative Parallel Computing (InPar)*. 1–11. <https://doi.org/10.1109/InPar.2012.6339606>
- [26] I-Jui Sung, Juan Gómez-Luna, José María González-Linares, Nicolás Guil, and Wen-Mei W. Hwu. 2014. In-place transposition of rectangular matrices on accelerators. *ACM SIGPLAN Notices* 49, 8 (Feb. 2014), 207–218. <https://doi.org/10.1145/2692916.2555266>
- [27] Andrey Vladimirov. 2013. *Multithreaded Transposition of Square Matrices with Common Code for Intel Xeon Processors and Intel Xeon Phi Coprocessors*. Technical Report.
- [28] F. Vázquez, J. J. Fernández, and E. M. Garzón. 2011. A new approach for sparse matrix vector product on NVIDIA GPUs. *Concurrency and Computation: Practice and Experience* 23, 8 (2011), 815–826. <https://doi.org/10.1002/cpe.1658> arXiv:<https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.1658>
- [29] Hao Wang, Weifeng Liu, Kaixi Hou, and Wu-chun Feng. 2016. Parallel Transposition of Sparse Data Structures. In *Proceedings of the 2016 International Conference on Supercomputing (ICS '16)*. Association for Computing Machinery, Istanbul, Turkey, 1–13. <https://doi.org/10.1145/2925426.2926291>
- [30] Tien-Hsiung Weng, Delgerdalai Batjargal, Hoa Pham, Meng-Yen Hsieh, and Kuan-Ching Li. 2013. Parallel Matrix Transposition and Vector Multiplication Using OpenMP. In *Intelligent Technologies and Engineering Systems (Lecture Notes in Electrical Engineering)*, Jengnan Juang and Yi-Cheng Huang (Eds.). Springer, New York, NY, 243–249. [https://doi.org/10.1007/978-1-4614-6747-2\\_30](https://doi.org/10.1007/978-1-4614-6747-2_30)
- [31] Tien-Hsiung Weng, Hoa Pham, Hai Jiang, and Kuan-Ching Li. 2013. Designing Parallel Sparse Matrix Transposition Algorithm Using CSR for GPUs. In *Intelligent Technologies and Engineering Systems (Lecture Notes in Electrical Engineering)*, Jengnan Juang and Yi-Cheng Huang (Eds.). Springer, New York, NY, 251–257. [https://doi.org/10.1007/978-1-4614-6747-2\\_31](https://doi.org/10.1007/978-1-4614-6747-2_31)
- [32] Albert-Jan N. Yzelman and Rob H. Bisseling. 2012. A Cache-Oblivious Sparse Matrix–Vector Multiplication Scheme Based on the Hilbert Curve. In *Progress in Industrial Mathematics at ECMI 2010 (Mathematics in Industry)*, Michael Günther, Andreas Bartel, Markus Brunk, Sebastian Schöps, and Michael Striebel (Eds.). Springer, Berlin, Heidelberg, 627–633. [https://doi.org/10.1007/978-3-642-25100-9\\_73](https://doi.org/10.1007/978-3-642-25100-9_73)
- [33] Keliang Zhang and Baifeng Wu. 2012. A Novel Parallel Approach of Radix Sort with Bucket Partition Preprocess. In *2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems*. 989–994. <https://doi.org/10.1109/HPCC.2012.144>

## A AGGREGATE RESULTS

These tables contain statistics about the performance of the algorithms across all permutations and tensors. In addition, we counted the number of times that each strategy was the best of all of the strategies. We exclude Top-2-sadilla and Top-3-sadilla from these results, as the strategy is not comparable across tensors of different orders.

**Table 2: Aggregate timing results (serial)**

stat	qsort	splatt	1-sadilla	2-sadilla	quesadilla	radix
min	0.61	1.00	0.31	0.22	0.00	0.51
Q1	1.91	1.00	1.34	0.91	0.54	1.00
median	2.32	1.00	1.54	1.19	0.84	1.41
Q3	3.18	1.00	1.83	1.43	1.27	2.26
max	6.36	1.00	3.91	3.91	5.78	7.84
wins	0	25	2.2	15	58	0.25

**Table 3: Aggregate timing results (parallel)**

stat	qsort	splatt	1-sadilla	2-sadilla	quesadilla	radix
min	1.27	1	0.27	0.063	0.00	0.13
Q1	4.40	1.00	0.99	0.62	0.64	1.25
median	20.38	1.00	1.06	1.08	1.25	2.22
Q3	28.29	1.00	1.24	1.49	2.09	3.84
max	86.70	1.00	2.02	5.69	6.82	9.83
wins	0	36	12	28	24	0

**Table 4: Median results by tensor (serial)**

filename	qsort	splatt	1-sadilla	2-sadilla	quesadilla	radix
flickr-3d	4.31	1.00	1.71	1.45	0.99	2.40
nell-1	3.04	1.00	1.94	1.15	0.81	3.08
nell-2	2.80	1.00	1.90	1.35	0.60	1.34
vast-2015-mc1-3d	3.49	1.00	1.49	1.18	0.55	2.07
chicago-crime-comm	2.18	1.00	1.71	1.08	0.52	0.93
delicious-4d	2.66	1.00	1.51	1.11	1.17	1.89
enron	2.43	1.00	1.53	1.06	0.64	1.18
flickr-4d	3.48	1.00	1.54	1.21	1.11	2.02
nips	2.84	1.00	1.88	1.35	0.81	1.79
uber	2.24	1.00	1.68	1.30	0.63	1.10
lbnl-network	2.29	1.00	1.67	1.34	1.04	1.37
vast-2015-mc1-5d	1.97	1.00	1.35	0.92	0.75	1.25

**Table 5: Median by tensor (parallel)**

filename	qsort	splatt	1-sadilla	2-sadilla	quesadilla	radix
flickr-3d	32.39	1.00	1.06	1.03	0.90	5.52
nell-1	22.94	1.00	1.08	1.05	0.88	6.33
nell-2	32.69	1.00	1.29	1.52	1.35	2.69
vast-2015-mc1-3d	25.64	1.00	1.30	0.93	0.60	2.45
chicago-crime-comm	19.41	1.00	1.25	0.93	0.84	1.82
delicious-4d	27.22	1.00	1.15	1.52	1.88	4.85
enron	28.81	1.00	1.28	1.68	1.61	3.20
flickr-4d	27.69	1.00	1.11	1.49	1.74	5.73
nips	40.06	1.00	1.43	1.37	1.24	2.02
uber	41.10	1.00	1.60	1.53	0.95	1.70
lbnl-network	3.97	1.00	1.01	0.68	1.01	1.33
vast-2015-mc1-5d	23.18	1.00	1.05	1.28	1.84	3.06

**Table 6: Wins by tensor (serial)**

filename	qsort	splatt	1-sadilla	2-sadilla	quesadilla	radix
flickr-3d	0%	50%	0%	0%	50%	0%
nell-1	0%	33.3%	0%	0%	66.7%	0%
nell-2	0%	33.3%	0%	0%	66.7%	0%
vast-2015-mc1-3d	0%	0%	0%	0%	100%	0%
chicago-crime-comm	0%	12.5%	0%	0%	87.5%	0%
delicious-4d	0%	45.8%	0%	16.7%	37.5%	0%
enron	0%	12.5%	0%	4.17%	79.2%	4.17%
flickr-4d	0%	45.8%	0%	12.5%	41.7%	0%
nips	0%	25%	12.5%	8.33%	54.2%	0%
uber	0%	16.7%	0%	0%	83.3%	0%
lbnl-network	0%	36.7%	0%	16.7%	46.7%	0%
vast-2015-mc1-5d	0%	10.8%	5%	25.8%	58.3%	0%

**Table 7: Wins by tensor (parallel)**

filename	qsort	splatt	1-sadilla	2-sadilla	quesadilla	radix
flickr-3d	0%	16.7%	16.7%	0%	66.7%	0%
nell-1	0%	16.7%	16.7%	16.7%	50%	0%
nell-2	0%	50%	16.7%	0%	33.3%	0%
vast-2015-mc1-3d	0%	33.3%	16.7%	0%	50%	0%
chicago-crime-comm	0%	8.33%	16.7%	20.8%	54.2%	0%
delicious-4d	0%	50%	16.7%	12.5%	20.8%	0%
enron	0%	79.2%	0%	0%	20.8%	0%
flickr-4d	0%	41.7%	16.7%	20.8%	20.8%	0%
nips	0%	62.5%	4.17%	8.33%	25%	0%
uber	0%	37.5%	0%	8.33%	54.2%	0%
lbnl-network	0%	11.7%	11.7%	53.3%	23.3%	0%
vast-2015-mc1-5d	0%	49.2%	15%	26.7%	9.17%	0%



## **B DETAILED RESULTS**

These tables contain the results of running all of the experiments. They are organized by file and the permutations are ordered lexicographically. A cell that contains a value of 1 is colored white. This value means that it performed as well as SPLATT. A cell that contains a value  $> 1$  is colored red and performed worse than SPLATT. A cell that contains a value  $< 1$  is colored blue and performed better than SPLATT.

**Figure 7: flickr-3d results normalized by splatt (serial)**

	qsort	splatt	1-sadilla	2-sadilla	quesadilla	radix
123	4.12	1	1.84	0.56	0	5.59
132	2.75	1	1.56	1.47	1.01	1.85
213	4.4	1	1.4	1.43	0.97	2.44
231	4.87	1	1.58	3.37	3.09	3.41
312	5.48	1	2.07	1.38	0.75	2.35
321	4.22	1	2	1.56	1.4	1.66

**Figure 8: nell-1 results normalized by splatt (serial)**

	qsort	splatt	1-sadilla	2-sadilla	quesadilla	radix
123	4.22	1	1.99	0.96	0	7.84
132	3.12	1	1.89	3.91	3.69	3.8
213	4.23	1	2.14	1.34	0.7	4.83
231	2.96	1	2.05	2.51	2.36	2.37
312	1.77	1	1.01	0.88	0.77	1.16
321	1.7	1	1.08	0.93	0.84	0.91

**Figure 9: nell-2 results normalized by splatt (serial)**

	qsort	splatt	1-sadilla	2-sadilla	quesadilla	radix
123	4.15	1	2.41	1.34	0	4.15
132	2.5	1	1.8	1.62	1.34	1.47
213	5.5	1	2.87	1.76	0.52	3.6
231	2.55	1	1.99	1.37	1.09	1.22
312	3.04	1	1.36	0.92	0.65	0.95
321	2.26	1	1.4	0.77	0.55	0.61

**Figure 10: vast-2015-mc1-3d results normalized by splatt (serial)**

	qsort	splatt	1-sadilla	2-sadilla	quesadilla	radix
123	4.72	1	1.54	0.38	0	4.68
132	4.05	1	1.41	2.13	0.84	3.22
213	3.35	1	1.44	0.72	0.62	1.14
231	2.72	1	1.2	1.47	0.72	0.82
312	3.63	1	3.91	1.55	0.48	3.01
321	1.95	1	2.08	0.88	0.41	0.61

**Figure 11: chicago-crime-comm results normalized by splatt (serial)**

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	quesadilla	radix
1234	3.2	1	1.43	0.95	0.38	0	3.07
1243	2.46	1	1.37	1.02	1.22	0.89	1.32
1324	2.17	1	1.23	1.01	0.82	0.61	1.62
1342	2.19	1	1.33	1.18	1.47	1.24	1.55
1423	2.08	1	1.31	0.86	0.62	0.38	1.1
1432	2.33	1	1.51	1.26	0.95	0.82	1.45
2134	3.79	1	2.15	1.09	0.67	0.38	2.24
2143	2.91	1	1.88	1.07	1.02	0.85	1.14
2314	1.77	1	1.67	0.82	0.4	0.28	0.73
2341	1.76	1	1.65	1.21	0.89	0.53	0.6
2413	1.94	1	1.75	0.85	0.4	0.24	0.57
2431	1.91	1	1.76	1.33	0.64	0.31	0.51
3124	4.38	1	2.07	0.99	0.71	0.39	2.17
3142	3.42	1	1.9	1.02	1.39	1.14	1.62
3214	2.15	1	1.63	1.03	0.6	0.54	0.75
3241	2.07	1	1.62	1.47	0.9	0.55	0.62
3412	2.19	1	1.76	1.13	0.68	0.53	0.77
3421	2.11	1	1.72	1.54	0.83	0.54	0.61
4123	4.7	1	2.22	1.13	0.74	0.35	1.73
4132	3.33	1	1.96	1.14	1.59	1.37	1.52
4213	2.28	1	1.7	0.94	0.44	0.26	0.62
4231	2.1	1	1.62	1.41	0.66	0.31	0.52
4312	2.15	1	1.77	0.79	0.37	0.25	0.69
4321	2.11	1	1.76	1.23	0.83	0.5	0.53

**Figure 12: delicious-4d results normalized by splatt (serial)**

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	quesadilla	radix
1234	3.51	1	1.43	0.47	0.31	0	5.44
1243	3.64	1	1.49	0.49	2.28	1.84	4.9
1324	2.61	1	1.49	1.54	1.31	1.15	2.36
1342	2.58	1	1.5	1.59	2.05	1.81	2.6
1423	2.58	1	1.3	1.26	1.07	0.77	2.93
1432	2.52	1	1.38	1.43	2.35	2.12	2.85
2134	2.68	1	1.35	1.1	0.99	0.87	2.07
2143	2.57	1	1.24	1.05	1.81	1.6	2.03
2314	2.22	1	1.29	1.38	1.29	1.18	1.7
2341	2.24	1	1.29	1.48	1.47	1.35	1.59
2413	2.23	1	1.24	0.88	0.85	0.65	1.45
2431	2.19	1	1.3	0.91	1.38	1.26	1.47
3124	3.95	1	1.58	0.9	0.71	0.55	2.27
3142	3.46	1	1.51	0.89	2.08	1.87	2.21
3214	2.85	1	1.67	1.3	1.22	1.1	1.44
3241	2.8	1	1.7	1.38	1.15	1.04	1.25
3412	2.5	1	1.53	0.84	0.67	0.54	1.62
3421	2.65	1	1.68	1.13	1.36	1.24	1.29
4123	4.8	1	2.05	0.99	0.73	0.38	4.06
4132	3.75	1	1.84	1	2.44	2.25	3.01
4213	2.67	1	1.81	1.26	1.22	0.95	1.73
4231	2.69	1	1.85	1.34	1.69	1.54	1.76
4312	2.54	1	1.81	0.95	0.76	0.61	1.74
4321	2.69	1	1.96	1.23	1.5	1.38	1.41

**Figure 13: enron results normalized by splatt (serial)**

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	quesadilla	radix
1234	3.06	1	1.96	1.3	0.37	0	2.83
1243	2.17	1	1.61	1.19	0.81	0.36	1.27
1324	1.88	1	1.38	1.04	0.81	0.65	1.18
1342	1.85	1	1.39	1.22	1.25	0.97	1.03
1423	1.91	1	1.52	0.82	0.63	0.25	1.01
1432	1.81	1	1.46	1.01	1.02	0.77	0.99
2134	4.08	1	2.02	1.64	0.73	0.39	2.62
2143	2.76	1	1.66	1.43	1.09	0.63	1.26
2314	2.3	1	1.54	1.25	1.15	0.92	1.25
2341	2.44	1	1.64	1.42	1.44	1.23	1.46
2413	2.39	1	1.65	0.77	0.74	0.3	1.01
2431	2.5	1	1.73	0.86	1.18	1.1	1.29
3124	3.11	1	1.43	1.04	0.72	0.48	1.22
3142	2.48	1	1.35	1.06	1.38	1.1	0.92
3214	2.15	1	1.31	0.85	0.78	0.59	0.78
3241	2.23	1	1.38	0.96	0.86	0.75	0.94
3412	2.01	1	1.34	0.82	0.8	0.5	0.79
3421	2.14	1	1.44	0.99	0.88	0.71	0.8
4123	5.53	1	2.23	1.77	1.3	0.39	2.46
4132	2.85	1	1.47	1.2	1.35	1.07	1.18
4213	3.56	1	1.77	1.15	1.1	0.47	1.43
4231	3.42	1	1.77	1.17	1.61	1.47	1.68
4312	2.42	1	1.46	0.9	0.86	0.54	0.89
4321	2.63	1	1.58	1.06	1.01	0.96	1.07

**Figure 14: flickr-4d results normalized by splatt (serial)**

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	quesadilla	radix
1234	3.45	1	1.47	0.47	0.28	0	4.93
1243	3.51	1	1.54	0.48	1.44	0.9	3.49
1324	2.44	1	1.4	1.27	0.99	0.89	1.61
1342	2.29	1	1.31	1.22	1.51	1.23	1.64
1423	3.33	1	1.47	1.76	1.18	0.78	4.45
1432	2.34	1	1.28	1.44	1.83	1.53	2.08
2134	4.21	1	1.28	1.32	1.12	0.92	2.53
2143	3.96	1	1.26	1.26	1.55	1.17	2.51
2314	4.01	1	1.3	2.74	2.74	2.56	3.13
2341	4.22	1	1.39	3.07	3.44	3.06	3.31
2413	3.9	1	1.22	1.15	1.21	0.85	2.02
2431	4.01	1	1.34	1.26	2.08	1.86	2.11
3124	4.63	1	1.72	1.19	0.79	0.61	2.02
3142	4.49	1	1.66	1.19	2.16	1.71	2.14
3214	3.66	1	1.75	1.31	1.36	1.19	1.59
3241	3.61	1	1.73	1.33	1.28	1.16	1.37
3412	3.06	1	1.54	1.03	0.94	0.62	1.51
3421	3.19	1	1.62	1.17	1.36	1.15	1.28
4123	4.68	1	2.3	1.27	0.78	0.38	3.93
4132	3.17	1	1.79	1.12	1.68	1.37	1.79
4213	2.72	1	1.96	0.97	0.91	0.73	1.49
4231	2.77	1	1.95	0.96	1.96	1.85	2.01
4312	2.36	1	1.77	0.85	0.77	0.53	1.24
4321	2.49	1	1.85	0.99	1.14	1.07	1.12

**Figure 15: nips results normalized by splatt (serial)**

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	quesadilla	radix
1234	3.88	1	1.89	1.65	0.34	0	3.78
1243	3.85	1	1.9	1.66	2.41	0.78	3.48
1324	0.85	1	0.44	0.76	0.73	0.66	0.88
1342	0.86	1	0.44	0.76	0.92	0.82	0.88
1423	3.79	1	1.88	2.6	2.37	0.77	3.52
1432	0.61	1	0.31	0.41	0.6	0.53	0.6
2134	5.67	1	2.49	2.21	0.96	0.62	4.01
2143	5.61	1	2.48	2.2	2.7	1.15	3.3
2314	3.68	1	1.89	2.05	2.04	1.6	2.53
2341	3.66	1	1.88	2.06	2.54	2.29	2.2
2413	5.49	1	2.42	2.51	2.44	0.91	3.33
2431	4.36	1	2.03	1.99	2.59	2.32	2.98
3124	2.89	1	1.35	0.82	0.73	0.57	1.25
3142	2.95	1	1.4	0.85	1.23	0.97	1.29
3214	2.42	1	1.58	0.86	0.84	0.71	0.92
3241	2.32	1	1.57	0.78	0.94	0.81	0.91
3412	2.79	1	1.28	1.34	1.09	0.87	1.17
3421	2.37	1	1.41	1.47	0.98	0.87	0.99
4123	3.44	1	3.14	2.15	1.89	0.51	3.18
4132	1.82	1	1.73	1.15	1.86	1.64	1.83
4213	2.55	1	2.41	1.36	1.28	0.48	1.75
4231	2.46	1	2.33	1.32	1.73	1.55	1.97
4312	1.88	1	1.73	0.85	0.67	0.5	0.88
4321	2.06	1	2	1.27	0.96	0.85	0.96

**Figure 16: uber results normalized by splatt (serial)**

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	quesadilla	radix
1234	3.41	1	2.19	1.5	0.62	0	2.83
1243	2.27	1	1.68	1.34	1.08	0.79	1.46
1324	1.95	1	1.44	1.06	0.79	0.46	1.5
1342	2.06	1	1.47	1.36	1.35	1.14	1.69
1423	1.95	1	1.52	0.77	0.6	0.34	1.12
1432	2.12	1	1.66	1.13	1.11	0.91	1.4
2134	4.36	1	2.79	1.85	1.03	0.46	2.54
2143	2.82	1	2.06	1.54	1.5	1.2	1.44
2314	1.93	1	1.73	1.01	0.63	0.37	1
2341	1.88	1	1.74	1.39	1.01	0.71	1.08
2413	1.94	1	1.78	0.77	0.48	0.26	0.75
2431	2.07	1	1.88	1.21	0.87	0.61	0.92
3124	3.96	1	2.02	1.33	0.88	0.46	2.25
3142	2.7	1	1.76	1.27	1.48	1.38	1.74
3214	2.22	1	1.56	1.24	0.89	0.64	0.85
3241	2.06	1	1.55	1.57	0.97	0.68	1.02
3412	1.9	1	1.59	1	0.72	0.55	0.97
3421	1.98	1	1.67	1.37	1.1	0.74	0.92
4123	4.82	1	2.04	1.34	0.93	0.46	2.3
4132	3.13	1	1.74	1.26	1.61	1.35	1.75
4213	2.6	1	1.61	1.26	0.9	0.65	0.85
4231	2.44	1	1.54	1.55	0.96	0.6	1.01
4312	2.26	1	1.61	0.99	0.72	0.55	0.96
4321	2.34	1	1.65	1.34	1.08	0.82	0.91

**Figure 17: lbnl-network results normalized by splatt (serial) (1)**

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	4-sadilla	quesadilla	radix
12345	2.01	1	1.42	1.33	1.21	1.14	0	1.46
12354	1.76	1	1.25	1.15	1.07	0.84	0.7	1.34
12435	1.65	1	1.14	1.08	1.26	1.24	0.4	1
12453	1.69	1	1.21	1.13	1.35	1.07	0.95	1.19
12534	2.12	1	1.5	1.4	0.93	0.87	0.7	1.48
12543	2.08	1	1.52	1.4	0.94	1.33	1.18	1.52
13245	1.46	1	1	1.21	1.16	1.09	0.33	0.97
13254	1.46	1	1.04	1.25	1.19	0.9	0.8	0.95
13425	1.44	1	0.99	1.2	1.37	1.32	0.58	0.96
13452	1.41	1	1.02	1.23	1.42	1.13	1.02	1.05
13524	1.84	1	1.31	1.61	1.04	1.05	0.99	1.35
13542	1.88	1	1.3	1.61	1.12	1.46	1.33	1.25
14235	1.51	1	1.06	1.19	1.12	1.11	0.35	0.89
14253	1.58	1	1.15	1.26	1.21	1	0.8	1.07
14325	1.49	1	1.04	1.17	1.31	1.28	0.56	0.93
14352	1.47	1	1.06	1.18	1.37	1.09	0.99	1.01
14523	1.93	1	1.39	1.51	1.14	1.14	1	1.33
14532	1.95	1	1.38	1.53	1.13	1.45	1.31	1.28
15234	2.28	1	1.66	0.95	0.95	0.95	0.79	1.59
15243	2.27	1	1.63	0.92	0.94	1.44	1.28	1.58
15324	2.25	1	1.63	0.95	1.31	1.29	1.14	1.53
15342	2.25	1	1.63	0.95	1.31	1.7	1.53	1.53
15423	2.26	1	1.64	0.95	1.35	1.36	1.15	1.58
15432	2.26	1	1.62	0.95	1.34	1.69	1.54	1.57

**Figure 18: lbnl-network results normalized by splatt (serial) (2)**

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	4-sadilla	quesadilla	radix
21345	2.58	1	1.85	1.64	1.53	1.43	0.35	1.41
21354	2.17	1	1.59	1.41	1.32	1.04	0.9	1.25
21435	2.21	1	1.56	1.41	1.47	1.47	0.58	1.14
21453	2.28	1	1.67	1.49	1.57	1.36	1.23	1.24
21534	2.67	1	1.97	1.74	1.21	1.21	1.05	1.44
21543	2.67	1	1.95	1.74	1.13	1.44	1.42	1.48
23145	1.81	1	1.43	1.21	1.18	1.11	0.37	0.95
23154	1.75	1	1.42	1.23	1.21	0.97	0.86	1
23415	1.72	1	1.34	1.18	1.24	1.23	0.51	0.92
23451	1.79	1	1.45	1.25	1.33	0.95	0.84	0.95
23514	2.12	1	1.72	1.46	0.88	0.87	0.81	1.2
23541	2.13	1	1.72	1.5	0.95	1.2	1.08	1.24
24135	1.74	1	1.4	1.15	1.12	1.08	0.37	0.92
24153	1.91	1	1.5	1.23	1.18	1.05	0.94	1.01
24315	1.71	1	1.38	1.16	1.21	1.2	0.49	0.89
24351	1.85	1	1.45	1.24	1.3	0.93	0.83	0.88
24513	2.25	1	1.81	1.54	0.99	0.99	0.86	1.22
24531	2.26	1	1.82	1.53	0.99	1.18	1.05	1.21
25134	2.56	1	2.06	0.9	0.9	0.9	0.75	1.41
25143	2.61	1	2.09	0.91	0.9	1.38	1.16	1.38
25314	2.56	1	2.13	0.91	1.12	1.13	0.98	1.4
25341	2.57	1	2.12	0.91	1.12	1.36	1.21	1.41
25413	2.56	1	2.1	0.9	1.2	1.2	1.04	1.45
25431	2.65	1	2.07	0.89	1.19	1.41	1.24	1.45

**Figure 19: lbnl-network results normalized by splatt (serial) (3)**

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	4-sadilla	quesadilla	radix
31245	2.68	1	1.74	1.62	1.55	1.44	0.36	1.35
31254	2.28	1	1.52	1.42	1.34	0.99	0.86	1.29
31425	2.05	1	1.35	1.26	1.46	1.41	0.56	1.08
31452	2.2	1	1.44	1.33	1.58	1.26	1.13	1.23
31524	2.67	1	1.81	1.66	1.15	1.15	1	1.53
31542	2.7	1	1.83	1.66	1.16	1.43	1.26	1.52
32145	2.06	1	1.46	1.42	1.35	1.29	0.43	1.09
32154	1.9	1	1.32	1.29	1.25	0.97	0.86	1.06
32415	1.79	1	1.28	1.23	1.31	1.3	0.54	0.96
32451	1.98	1	1.37	1.33	1.41	0.93	0.89	1.06
32514	2.31	1	1.66	1.59	0.93	0.93	0.87	1.24
32541	2.34	1	1.64	1.6	1	1.27	1.14	1.2
34125	1.84	1	1.26	1.24	1.17	1.13	0.39	0.96
34152	1.91	1	1.33	1.35	1.28	1.01	0.9	1.09
34215	1.66	1	1.19	1.23	1.21	1.2	0.5	0.89
34251	1.79	1	1.29	1.29	1.27	0.89	0.83	0.98
34512	2.34	1	1.65	1.71	1.03	1.03	0.89	1.28
34521	2.32	1	1.66	1.73	1.02	1.28	1.14	1.3
35124	2.79	1	1.96	0.96	0.97	0.96	0.81	1.54
35142	2.72	1	1.98	0.97	0.97	1.43	1.27	1.51
35214	2.69	1	1.97	0.9	1.21	1.27	1.11	1.57
35241	2.77	1	2	0.97	1.26	1.52	1.36	1.42
35412	2.81	1	1.97	0.97	1.28	1.28	1.13	1.43
35421	2.74	1	1.96	0.9	1.27	1.5	1.35	1.56

**Figure 20: lbnl-network results normalized by splatt (serial) (4)**

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	4-sadilla	quesadilla	radix
41235	2.82	1	1.88	1.62	1.51	1.48	0.36	1.44
41253	2.86	1	1.9	1.66	1.54	1.34	1.11	1.53
41325	2.29	1	1.56	1.33	1.58	1.54	0.62	1.15
41352	2.31	1	1.6	1.4	1.69	1.4	1.28	1.28
41523	2.99	1	2.03	1.76	1.38	1.36	1.21	1.65
41532	3	1	2.05	1.79	1.38	1.84	1.66	1.59
42135	1.81	1	1.37	1.15	1.1	1.1	0.37	0.91
42153	1.96	1	1.51	1.23	1.19	1.04	0.94	1.01
42315	1.79	1	1.36	1.16	1.22	1.22	0.5	0.89
42351	1.86	1	1.48	1.26	1.32	0.94	0.84	1
42513	2.41	1	1.89	1.6	1.03	1.03	0.9	1.28
42531	2.44	1	1.93	1.6	1.02	1.23	1.11	1.28
43125	1.68	1	1.33	1.13	1.07	1.04	0.35	0.87
43152	1.78	1	1.4	1.19	1.14	0.95	0.85	0.97
43215	1.74	1	1.36	1.24	1.18	1.22	0.51	0.91
43251	1.91	1	1.5	1.35	1.35	0.88	0.8	1
43512	2.25	1	1.74	1.6	0.96	0.96	0.83	1.2
43521	2.25	1	1.73	1.57	0.95	1.19	1.07	1.23
45123	2.82	1	2.17	0.97	0.92	0.97	0.82	1.42
45132	2.75	1	2.2	0.97	0.97	1.46	1.31	1.49
45213	2.76	1	2.21	0.94	1.16	1.28	1.1	1.55
45231	2.83	1	2.18	0.98	1.27	1.5	1.34	1.54
45312	2.86	1	2.16	0.98	1.22	1.21	1.06	1.51
45321	2.76	1	2.17	0.97	1.21	1.41	1.26	1.49



**Figure 21: lbnl-network results normalized by splatt (serial) (5)**

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	4-sadilla	quesadilla	radix
51234	6.09	1	1.77	1.66	1.65	1.66	1.26	3.29
51243	6.04	1	1.67	1.58	1.66	3.16	2.67	3.27
51324	6.05	1	1.75	1.64	3.05	3.02	2.57	3.2
51342	6.29	1	1.86	1.73	3.23	4.63	4.19	3.06
51423	6.06	1	1.76	1.64	3.11	3.11	2.63	3.19
51432	6.36	1	1.87	1.74	3	4.39	4.2	3.4
52134	6.06	1	1.76	2.26	2.24	2.26	1.87	3.29
52143	6.04	1	1.78	2.28	2.27	3.67	3.3	3.34
52314	5.92	1	1.79	2.28	2.75	2.75	2.34	3.31
52341	5.97	1	1.78	2.27	2.75	3.28	2.86	3.28
52413	5.99	1	1.76	2.25	2.77	2.77	2.38	3.27
52431	6.02	1	1.77	2.27	2.83	3.26	2.88	3.3
53124	6.2	1	1.83	2.22	2.2	2.21	1.8	3.33
53142	6.27	1	1.86	2.21	2.23	3.66	3.27	3.31
53214	6.29	1	1.84	2.01	2.5	2.7	2.29	3.05
53241	6.27	1	1.85	2.21	2.72	3.27	2.85	2.98
53412	6.22	1	1.83	2.19	2.75	2.61	2.33	3.31
53421	6.11	1	1.84	2.2	2.75	3.24	2.89	3.33
54123	6.21	1	1.82	2.31	2.32	2.16	1.79	3.41
54132	6.09	1	1.84	2.35	2.35	3.7	3.29	3.38
54213	6.15	1	1.82	2.34	2.84	2.82	2.46	3.35
54231	6.2	1	1.86	2.39	2.9	3.4	3.01	3.44
54312	6.28	1	1.7	2.36	2.83	2.81	2.43	3.42
54321	6.02	1	1.74	2.33	2.78	3.14	2.91	3.37

**Figure 22: vast-2015-mc1-5d results normalized by splatt (serial) (1)**

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	4-sadilla	quesadilla	radix
12345	3.38	1	1.07	0.22	0.22	0.22	0	4.24
12354	3.41	1	1.1	0.23	0.23	2.99	2.63	4.13
12435	3.39	1	1.08	0.23	2.99	2.76	2.79	4.44
12453	3.35	1	1.07	0.22	2.86	6.36	5.57	4.98
12534	3.36	1	1.08	0.23	2.82	3.01	2.72	4.39
12543	3.41	1	1.08	0.23	3.09	5.96	5.78	4.75
13245	2.82	1	0.92	1.44	0.81	0.76	0.62	3.56
13254	2.81	1	0.93	1.49	0.81	3.24	3.16	3.56
13425	2.11	1	0.97	1.28	1.35	1.3	1.12	2.23
13452	2.01	1	0.98	1.25	1.41	2.29	2.21	1.9
13524	2.19	1	0.98	1.3	1.31	1.33	1.08	2.28
13542	2.1	1	0.96	1.22	1.4	2.3	2.18	1.78
14235	2.26	1	1.05	1.29	1.08	1.17	0.98	2.32
14253	2.37	1	1.06	1.26	1.14	2.83	2.59	2.84
14325	2.32	1	1.04	1.28	1.78	1.83	1.51	2.28
14352	2.2	1	1.02	1.21	1.91	2.56	2.29	2.09
14523	2.19	1	1.02	1.3	2.3	2.27	2	2.36
14532	2.19	1	1.01	1.29	2.24	2.94	2.63	1.9
15234	2.38	1	1.06	1.29	1.16	1.18	0.98	2.46
15243	2.4	1	1.08	1.31	1.11	2.59	2.73	2.88
15324	2.26	1	1.04	1.29	1.86	1.86	1.63	2.46
15342	2.19	1	1.02	1.21	1.92	2.48	2.31	2.11
15423	2.22	1	1.02	1.31	2.39	2.28	2.16	2.15
15432	2.23	1	1.02	1.31	2.32	2.84	2.65	2.15

**Figure 23: vast-2015-mc1-5d results normalized by splatt (serial) (2)**

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	4-sadilla	quesadilla	radix
21345	2.32	1	0.99	0.49	0.53	0.52	0.44	1.54
21354	2.34	1	0.96	0.5	0.52	1.78	1.65	1.46
21435	2.43	1	0.97	0.54	1.74	1.86	1.67	1.5
21453	2.3	1	0.99	0.51	1.71	2.93	3.03	1.83
21534	2.23	1	0.98	0.51	1.67	1.82	1.66	1.45
21543	2.41	1	0.99	0.54	1.79	3.02	3.15	1.73
23145	1.95	1	0.81	1.03	0.61	0.63	0.59	1.17
23154	1.93	1	0.8	1.03	0.59	1.65	1.51	1.24
23415	1.52	1	0.91	1.05	0.65	0.56	0.51	1.09
23451	1.52	1	0.91	1.01	0.79	0.67	0.57	0.71
23514	1.48	1	0.85	0.98	0.65	0.56	0.47	1
23541	1.48	1	0.88	0.98	0.79	0.68	0.55	0.71
24135	1.58	1	0.97	0.69	0.53	0.54	0.47	1.16
24153	1.67	1	0.96	0.69	0.53	1.38	1.24	1.28
24315	1.51	1	0.93	0.65	0.9	0.76	0.69	1.14
24351	1.52	1	0.95	0.77	0.95	0.67	0.56	0.81
24513	1.52	1	0.96	0.77	0.65	0.61	0.55	0.77
24531	1.51	1	0.97	0.78	0.61	0.81	0.78	0.82
25134	1.56	1	0.93	0.65	0.55	0.55	0.49	1.08
25143	1.67	1	0.95	0.69	0.52	1.33	1.18	1.38
25314	1.52	1	0.91	0.68	0.93	0.72	0.7	1.04
25341	1.5	1	0.92	0.76	0.95	0.68	0.56	0.82
25413	1.52	1	0.92	0.82	0.65	0.61	0.5	0.84
25431	1.56	1	0.91	0.8	0.62	0.8	0.7	0.83

**Figure 24: vast-2015-mc1-5d results normalized by splatt (serial) (3)**

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	4-sadilla	quesadilla	radix
31245	2.59	1	2.88	1.15	0.54	0.54	0.38	2.94
31254	2.64	1	2.86	1.14	0.55	2.91	2.76	3.22
31425	1.79	1	1.95	0.96	1.3	1.23	1.15	1.77
31452	1.74	1	1.87	0.96	1.32	2.1	1.91	1.5
31524	1.81	1	1.89	0.95	1.36	1.23	1.13	1.86
31542	1.72	1	1.86	0.93	1.29	2.03	1.91	1.61
32145	1.42	1	1.49	0.67	0.38	0.39	0.31	0.94
32154	1.48	1	1.47	0.65	0.37	1.14	0.99	0.84
32415	1.34	1	1.42	0.87	0.61	0.51	0.43	1
32451	1.39	1	1.39	0.87	0.74	0.61	0.52	0.67
32514	1.35	1	1.41	0.84	0.63	0.47	0.47	0.94
32541	1.36	1	1.38	0.81	0.73	0.6	0.47	0.64
34125	1.48	1	1.5	0.68	0.28	0.24	0.19	0.95
34152	1.42	1	1.51	0.71	0.31	0.62	0.53	1.02
34215	1.47	1	1.58	1.1	0.63	0.52	0.5	0.89
34251	1.47	1	1.47	1.06	0.72	0.63	0.57	0.7
34512	1.25	1	1.36	0.95	0.56	0.28	0.21	0.74
34521	1.41	1	1.44	1.06	0.86	0.58	0.49	0.62
35124	1.47	1	1.51	0.67	0.28	0.23	0.18	0.98
35142	1.38	1	1.49	0.7	0.31	0.56	0.5	1.01
35214	1.46	1	1.57	1.12	0.61	0.53	0.5	0.91
35241	1.49	1	1.54	1.07	0.73	0.65	0.6	0.72
35412	1.28	1	1.34	0.94	0.56	0.28	0.2	0.74
35421	1.38	1	1.42	1.06	0.87	0.57	0.45	0.62

**Figure 25: vast-2015-mc1-5d results normalized by splatt (serial) (4)**

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	4-sadilla	quesadilla	radix
41235	5.3	1	2.28	0.86	0.66	0.68	0.47	3.38
41253	5.02	1	2.22	0.83	0.65	1.98	1.74	4.15
41325	4.9	1	2.15	0.8	1.16	0.98	0.84	3.26
41352	3.82	1	1.87	0.72	1.02	1.8	1.47	2.63
41523	4.02	1	1.89	0.75	1.49	1.53	1.28	3.25
41532	3.89	1	1.85	0.71	1.42	1.79	1.46	2.72
42135	2.19	1	1.57	0.82	0.72	0.69	0.59	1.34
42153	2.11	1	1.56	0.87	0.72	1.17	1.21	1.53
42315	2.05	1	1.45	0.79	0.98	0.86	0.8	1.25
42351	1.92	1	1.48	0.9	1.14	0.93	0.79	1.03
42513	2.04	1	1.47	0.97	0.8	0.76	0.69	1.07
42531	1.97	1	1.49	0.93	0.88	1.09	0.93	1.06
43125	3.44	1	1.53	1.99	0.97	0.86	0.81	2.41
43152	2.96	1	1.46	1.8	0.96	1.48	1.41	1.98
43215	1.86	1	1.33	1.57	0.8	0.62	0.57	1.1
43251	1.85	1	1.31	1.49	0.91	0.82	0.7	0.86
43512	1.67	1	1.22	1.4	0.71	0.35	0.27	0.9
43521	1.7	1	1.27	1.45	1.08	0.67	0.6	0.73
45123	1.91	1	1.43	0.78	0.36	0.33	0.25	1.28
45132	1.96	1	1.42	0.79	0.36	0.49	0.4	1.14
45213	1.96	1	1.5	1.2	0.76	0.67	0.65	0.85
45231	1.9	1	1.44	1.2	0.69	0.93	0.78	0.86
45312	1.69	1	1.25	0.67	0.9	0.49	0.45	0.95
45321	1.65	1	1.33	1.05	1.24	0.71	0.61	0.68

**Figure 26: vast-2015-mc1-5d results normalized by splatt (serial) (5)**

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	4-sadilla	quesadilla	radix
51234	5.17	1	2.26	0.82	0.66	0.67	0.46	3.65
51243	5.08	1	2.24	0.83	0.65	1.99	1.78	3.76
51324	5.03	1	2.24	0.81	1.19	1.05	0.8	3.48
51342	3.88	1	1.89	0.71	1.02	1.78	1.44	2.64
51423	3.94	1	1.84	0.73	1.42	1.37	1.32	2.85
51432	3.91	1	1.83	0.72	1.5	1.78	1.56	2.73
52134	2.22	1	1.53	0.84	0.72	0.66	0.66	1.32
52143	2.1	1	1.55	0.8	0.68	1.19	1.11	1.51
52314	2.08	1	1.47	0.84	1.01	0.83	0.81	1.25
52341	1.94	1	1.47	0.93	1.12	0.88	0.75	1.06
52413	2.05	1	1.49	0.97	0.87	0.86	0.76	1.09
52431	1.99	1	1.45	0.92	0.82	1.04	0.9	1
53124	3.45	1	1.51	1.99	1.05	0.94	0.71	2.36
53142	2.98	1	1.47	1.82	0.96	1.54	1.41	2.17
53214	1.89	1	1.37	1.54	0.81	0.62	0.61	1.11
53241	1.86	1	1.3	1.52	0.88	0.77	0.7	0.93
53412	1.67	1	1.21	1.39	0.71	0.34	0.26	0.92
53421	1.71	1	1.27	1.42	1.08	0.7	0.55	0.71
54123	2	1	1.39	0.8	0.35	0.33	0.24	1.23
54132	1.95	1	1.4	0.76	0.35	0.48	0.4	1.17
54213	1.98	1	1.48	1.21	0.76	0.66	0.65	0.87
54231	1.89	1	1.45	1.19	0.75	0.9	0.78	0.83
54312	1.71	1	1.25	0.66	0.92	0.53	0.44	0.94
54321	1.71	1	1.33	1.06	1.23	0.67	0.57	0.77

**Figure 27: flickr-3d results normalized by splatt (parallel)**

	qsort	splatt	1-sadilla	2-sadilla	k-sadilla	radix
123	30.16	1	0.29	0.27	0	9.83
132	34.61	1	0.56	2.06	1.82	7.27
213	5.23	1	1.08	0.63	0.6	0.96
231	5.27	1	1.05	0.96	0.91	1.02
312	40.67	1	1.27	1.1	0.9	5.86
321	36.65	1	1.3	4.52	4.37	5.18

**Figure 28: nell-1 results normalized by splatt (parallel)**

	qsort	splatt	1-sadilla	2-sadilla	k-sadilla	radix
123	19.94	1	0.35	0.16	0	7.08
132	25.94	1	0.46	3.69	3.5	6.4
213	33.9	1	1.25	1.16	0.91	7.54
231	34.15	1	1.34	5.69	5.41	6.26
312	8.07	1	1.07	0.75	0.76	1.16
321	8.18	1	1.09	0.93	0.86	1.08

**Figure 29: nell-2 results normalized by splatt (parallel)**

	qsort	splatt	1-sadilla	2-sadilla	k-sadilla	radix
123	28.95	1	0.39	0.24	0	4.48
132	30.24	1	0.74	2.13	1.86	2.93
213	38.84	1	1.26	1.13	0.8	3.88
231	28.38	1	1.37	1.98	1.74	2.31
312	36.86	1	1.33	1.35	1.16	2.46
321	35.15	1	1.41	1.69	1.54	2.09

**Figure 30: vast-2015-mc1-3d results normalized by splatt (parallel)**

	qsort	splatt	1-sadilla	2-sadilla	k-sadilla	radix
123	25.76	1	0.31	0.41	0	4.63
132	25.51	1	0.38	1.68	1.49	4.22
213	35.88	1	1.31	1.46	1.1	2.59
231	33.91	1	1.39	1.92	1.66	2.3
312	3.62	1	1.29	0.14	0.11	0.59
321	2.04	1	1.47	0.11	$9.8 \cdot 10^{-2}$	0.13

**Figure 31: chicago-crime-comm results normalized by splatt (parallel)**

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	k-sadilla	radix
1234	20.77	1	0.33	0.33	0.41	0	4.34
1243	22.91	1	0.48	0.44	1.9	1.63	4.05
1324	22.96	1	0.54	1.75	1.73	1.33	3.88
1342	23.23	1	0.53	1.71	3.02	2.46	3.72
1423	23.05	1	0.56	1.42	1.47	1.17	3.52
1432	26.74	1	0.58	1.7	2.97	2.59	3.56
2134	22.26	1	1.24	0.86	0.94	0.6	3.17
2143	19.48	1	1.22	0.77	1.72	1.43	2.39
2314	13.32	1	1.26	0.84	0.89	0.76	1.51
2341	13.16	1	1.3	0.95	1.11	0.97	1.29
2413	15.66	1	1.31	0.93	0.92	0.71	1.77
2431	15.23	1	1.36	1.03	1.17	0.99	1.32
3124	28.37	1	1.14	1.03	1.1	0.68	3.2
3142	23.35	1	1.2	0.93	2.08	1.77	2.58
3214	17.08	1	1.16	1	1.03	0.78	1.67
3241	19.92	1	1.39	1.43	1.47	1.29	1.71
3412	19.35	1	1.33	1.1	1.08	0.9	1.87
3421	19.86	1	1.34	1.36	1.36	1.17	1.53
4123	18.49	1	1.23	0.77	0.64	0.39	2.05
4132	10.97	1	1.31	0.57	0.78	0.64	1.12
4213	9.83	1	1.38	0.64	0.55	0.43	0.98
4231	8.49	1	1.29	0.81	0.62	0.5	0.69
4312	8.79	1	1.32	0.52	0.46	0.37	0.8
4321	9.14	1	1.37	0.81	0.62	0.52	0.69

**Figure 32: delicious-4d results normalized by splatt (parallel)**

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	k-sadilla	radix
1234	22	1	0.27	0.24	0.29	0	8.43
1243	22.77	1	0.29	0.24	2.22	1.98	8.09
1324	27.1	1	0.45	2.09	2.1	1.87	6.87
1342	28.57	1	0.48	2.17	3.67	3.45	6.94
1423	25.41	1	0.43	1.79	1.81	1.59	7.22
1432	26.43	1	0.46	1.88	3.77	3.52	7.04
2134	7.87	1	1.02	0.67	0.73	0.64	1.61
2143	7.85	1	1.04	0.69	1.05	1.08	1.54
2314	8.32	1	1.1	1.25	1.2	1.16	1.63
2341	8.17	1	1.01	1.21	1.33	1.29	1.51
2413	8.07	1	1.04	0.87	0.85	0.81	1.46
2431	7.82	1	1.04	0.89	1.28	1.23	1.42
3124	27.34	1	1.19	1.01	0.96	0.79	4.37
3142	29.79	1	1.22	1.25	2.6	2.41	4.5
3214	29.22	1	1.31	2.89	2.82	2.64	4.1
3241	29.22	1	1.29	3.02	3.4	3.19	3.8
3412	28.52	1	1.34	2.65	1.69	1.46	4.32
3421	27.02	1	1.24	3.7	3.24	2.98	3.67
4123	45.13	1	1.39	1.23	1.23	0.95	9.09
4132	42.01	1	1.3	1.14	3.94	3.62	7.74
4213	35.6	1	1.31	3.59	3.61	3.36	6.29
4231	36.14	1	1.42	3.67	5.61	5.27	6.03
4312	36.13	1	1.31	2	2.21	1.89	5.58
4321	38.52	1	1.35	2.17	4.62	4.48	5.2

**Figure 33: enron results normalized by splatt (parallel)**

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	k-sadilla	radix
1234	16.97	1	0.74	0.63	0.28	0	3.35
1243	11.96	1	1.1	1	0.81	0.62	1.62
1324	12.33	1	1.03	1.31	1.22	1.04	1.84
1342	11.91	1	1.06	1.42	2.02	1.92	1.64
1423	11.06	1	1.15	0.88	0.8	0.62	1.37
1432	10.94	1	1.13	1.28	1.46	1.39	1.33
2134	34.84	1	1.26	1.38	1.28	0.9	4.46
2143	35.46	1	1.4	1.64	2.39	2.08	3.5
2314	28.28	1	1.32	2.39	2.38	2.08	3.81
2341	29.33	1	1.3	2.58	3.55	3.14	4.01
2413	31.84	1	1.35	1.73	1.67	1.41	3.19
2431	33.92	1	1.35	1.97	3.22	2.91	3.68
3124	29.62	1	1.24	1.23	1.23	1	3.25
3142	27.79	1	1.28	1.36	2.34	2.16	2.84
3214	26.36	1	1.26	1.97	1.87	1.58	2.86
3241	27.02	1	1.28	2.13	2.57	2.32	3.01
3412	24.39	1	1.24	1.72	1.57	1.42	2.47
3421	26.19	1	1.27	1.98	2.47	2.15	2.85
4123	45.37	1	1.34	1.26	1.12	0.9	4.47
4132	35.29	1	1.39	1.24	2.48	2.25	3.45
4213	38.31	1	1.39	1.83	1.81	1.59	3.56
4231	38.51	1	1.35	1.96	3.27	3.02	3.74
4312	32.38	1	1.31	1.8	1.79	1.64	2.9
4321	34.32	1	1.31	2.01	2.87	2.57	3.2

**Figure 34: flickr-4d results normalized by splatt (parallel)**

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	k-sadilla	radix
1234	22.94	1	0.27	0.24	0.3	0	9.65
1243	23.13	1	0.28	0.25	2.09	1.82	8.61
1324	26.8	1	0.47	1.93	1.97	1.7	7
1342	27.1	1	0.52	1.9	3.38	3.17	7.13
1423	23.8	1	0.31	2.01	2.02	1.79	9.63
1432	24.55	1	0.47	1.87	3.57	3.38	7.58
2134	4.42	1	1.03	0.59	0.57	0.56	1.03
2143	4.51	1	1.03	0.58	0.77	0.75	1.02
2314	4.48	1	1.03	0.88	0.91	0.88	1.13
2341	4.4	1	1.02	0.91	1.04	1.02	1.14
2413	4.44	1	1.06	0.68	0.67	0.64	1.01
2431	4.28	1	1.03	0.68	0.87	0.85	0.97
3124	33.11	1	1.22	1.12	1.13	0.9	5.69
3142	31.63	1	1.17	1.09	2.5	2.29	5.69
3214	29.92	1	1.27	4.04	4.22	4.04	5.43
3241	29.43	1	1.26	4.1	4.63	4.48	5.08
3412	28.29	1	1.33	2.16	1.63	1.45	5.11
3421	29	1	1.4	2.65	4.6	4.4	5.26
4123	36.73	1	1.23	1.12	1.1	0.86	9.06
4132	37.87	1	1.33	1.16	3.03	3	7.72
4213	31.15	1	1.32	4.84	4.7	4.5	6.98
4231	31.86	1	1.32	4.81	7.08	6.82	7.58
4312	32.04	1	1.32	1.83	1.82	1.64	5.99
4321	31.65	1	1.28	1.82	5.27	5.09	5.76

**Figure 35: nips results normalized by splatt (parallel)**

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	k-sadilla	radix
1234	52.96	1	0.78	0.67	0.43	0	5.75
1243	50.01	1	0.84	0.76	3.07	2.43	5.78
1324	16.71	1	1.12	1.18	1.13	1.06	1.63
1342	16.79	1	1.13	1.17	1.93	1.91	1.71
1423	50.82	1	0.88	3.45	3.23	2.45	5.76
1432	12.27	1	1.64	1.44	1.44	1.3	1.1
2134	68.12	1	1.41	1.21	1.07	0.91	4.4
2143	66.32	1	1.59	1.39	3.34	2.67	4.62
2314	50.48	1	1.47	3.15	3.09	2.85	3.35
2341	50.55	1	1.33	3.17	3.32	3	3.61
2413	66.56	1	1.59	2.12	2.03	1.07	4.34
2431	56.64	1	1.59	1.9	4.11	3.22	3.81
3124	40.3	1	1.43	1.29	1.39	1.19	2.08
3142	39.82	1	1.36	1.36	2.65	2.27	2.03
3214	40.35	1	1.6	1.44	1.41	1.1	1.76
3241	39.29	1	1.39	1.49	1.59	1.42	1.71
3412	41.58	1	1.59	1.57	1.55	1.13	2.01
3421	38.93	1	1.42	1.53	1.74	1.47	1.89
4123	25.39	1	1.84	0.71	0.66	0.57	2.74
4132	14.84	1	1.92	1.44	1.12	1.04	1.39
4213	16.52	1	1.39	0.56	0.58	0.4	1.28
4231	17.58	1	1.56	0.67	1.39	1.3	1.37
4312	16.33	1	1.66	0.78	0.8	0.67	0.95
4321	15.33	1	1.35	0.78	0.84	0.66	0.77

**Figure 36: uber results normalized by splatt (parallel)**

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	k-sadilla	radix
1234	54.22	1	0.93	0.66	0.55	0	3.82
1243	44.93	1	1.1	0.79	1.77	1.61	2.06
1324	39.21	1	1.12	2.09	1.96	1.61	2.16
1342	41.63	1	1.09	2.09	3.22	3.01	2
1423	40.58	1	1.13	1.56	1.57	1.21	1.67
1432	45.75	1	1.13	2.09	2.77	2.63	1.73
2134	37.23	1	1.55	0.71	0.63	0.29	1.8
2143	30.16	1	2.02	0.69	1.17	1.18	1.15
2314	18.64	1	1.51	0.7	0.56	0.52	0.91
2341	18.12	1	1.51	1.04	0.81	0.69	0.72
2413	18.28	1	1.47	0.56	0.42	0.38	0.68
2431	19.51	1	1.42	0.94	0.78	0.59	0.61
3124	59.82	1	1.71	2.09	1.56	0.77	2.55
3142	46.19	1	1.68	2.2	2.04	2.07	1.74
3214	40.56	1	1.63	1.47	0.94	0.67	1.77
3241	38.01	1	1.57	2.44	1.61	1.35	1.35
3412	37.55	1	1.64	1.04	0.86	0.83	1.26
3421	38.2	1	1.66	1.37	1.26	0.97	1.17
4123	86.7	1	1.95	3.06	1.91	0.83	3.08
4132	63.18	1	1.94	4.09	2.56	2.4	2
4213	54.58	1	1.73	2.54	1.23	0.77	2
4231	52.89	1	1.82	4.87	1.98	1.52	1.58
4312	49.2	1	1.65	1.51	1.04	0.93	1.32
4321	50.71	1	1.76	2.58	1.57	1.09	1.3



Figure 37: lbnl-network results normalized by splatt (parallel) (1)

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	4-sadilla	k-sadilla	radix
12345	5.6	1	0.9	0.71	0.61	0.37	0	2.57
12354	4.12	1	0.84	0.7	0.48	1.2	1.06	1.88
12435	3.97	1	0.92	0.73	0.67	0.66	0.34	1.55
12453	3.46	1	0.74	0.6	0.54	1.39	1.36	1.46
12534	3.66	1	0.75	0.51	1.07	1.13	0.98	1.5
12543	4.19	1	0.9	0.65	1.16	1.66	1.62	1.87
13245	4.51	1	0.97	0.99	0.92	0.72	0.44	1.76
13254	4.33	1	1.09	1.26	0.92	1.44	1.5	1.86
13425	4.56	1	0.96	1.21	1.24	1.09	0.8	1.93
13452	4.13	1	0.73	1.08	1.1	2.03	2.13	1.78
13524	3.88	1	0.87	0.86	1.62	1.58	1.62	1.66
13542	4.6	1	0.83	0.99	1.68	2.25	2.24	1.96
14235	3.97	1	0.78	0.71	0.67	0.64	0.35	1.58
14253	4.34	1	0.81	0.75	0.68	1.51	1.39	1.82
14325	4.15	1	0.77	0.77	0.97	1.01	0.7	1.52
14352	4.54	1	0.82	0.9	1.1	2.1	2.06	1.74
14523	5.55	1	0.88	0.89	1.98	2.02	2.16	2.18
14532	5.74	1	0.93	0.91	2.31	2.71	2.73	2.23
15234	5.7	1	0.72	1.49	1.61	1.63	1.42	2.25
15243	5.97	1	0.97	1.55	1.59	1.85	2.02	2.35
15324	5.62	1	0.79	1.64	2.16	2.12	2.05	2.3
15342	6.28	1	1.01	1.9	2.52	2.95	2.83	2.43
15423	6.28	1	0.9	1.85	2.13	2.39	2.16	2.57
15432	5.65	1	0.74	1.57	2.02	2.64	2.48	2.19

Figure 38: lbnl-network results normalized by splatt (parallel) (2)

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	4-sadilla	k-sadilla	radix
21345	4.86	1	1.14	0.85	0.48	0.37	$5 \cdot 10^{-2}$	1.74
21354	3.73	1	1.19	0.81	0.43	0.82	0.79	1.24
21435	3.81	1	1.02	0.83	0.51	0.5	0.3	1.32
21453	3.4	1	0.89	0.75	0.45	1.01	1	1.02
21534	3.89	1	1.1	0.76	0.91	0.83	0.79	1.26
21543	3.76	1	1.22	0.88	0.88	1.16	1.09	1.25
23145	3.29	1	1.11	0.4	0.31	0.24	$7.2 \cdot 10^{-2}$	1.1
23154	3.11	1	1.21	0.58	0.35	0.71	0.64	1.11
23415	2.98	1	1.01	0.47	0.31	0.26	0.14	0.94
23451	3.09	1	1.16	0.51	0.28	0.98	0.96	0.99
23514	3.23	1	1.01	0.48	1.06	1.16	1.01	1.11
23541	3.15	1	1.03	0.47	1.12	1.02	1.02	1.06
24135	3.2	1	1.1	0.21	0.19	0.17	$7.4 \cdot 10^{-2}$	1.08
24153	3.35	1	1.17	0.2	0.18	0.71	0.64	1.09
24315	3.26	1	1.07	0.26	0.31	0.22	0.14	1.08
24351	3.3	1	1.12	0.24	0.24	1.05	1.09	1.05
24513	3.65	1	1.23	0.23	1.18	1.12	1.06	1.19
24531	3.56	1	1.17	0.22	1.07	1.1	1.08	1.16
25134	3.18	1	0.9	0.9	0.96	1.01	0.92	1.02
25143	3.67	1	1.15	1.03	1.1	1.17	1.2	1.16
25314	4.01	1	1.29	1.17	1.33	1.3	1.25	1.37
25341	3.2	1	0.79	1	1.07	1.03	0.96	1.13
25413	3.89	1	1.22	1.18	1.32	1.21	1.2	1.32
25431	4.02	1	1.09	1.09	1.24	1.15	1.27	1.3

**Figure 39: lbnl-network results normalized by splatt (parallel) (3)**

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	4-sadilla	k-sadilla	radix
31245	9.72	1	0.95	0.83	0.77	0.61	0.1	3.21
31254	6.56	1	0.91	0.9	0.57	1.34	1.45	2.05
31425	6.26	1	1.03	0.84	0.82	0.68	0.44	2.06
31452	7.26	1	1.02	0.88	1.02	1.91	2.02	2.45
31524	6.41	1	0.85	0.61	1.38	1.27	1.25	2.21
31542	6.53	1	0.96	0.62	1.47	1.64	1.72	2.27
32145	5.89	1	0.86	0.58	0.5	0.43	0.13	1.85
32154	6.08	1	1.03	0.79	0.59	1.24	1.3	1.98
32415	5.77	1	1.02	0.82	0.67	0.49	0.26	1.98
32451	6.62	1	1.05	0.75	0.6	1.95	1.99	2.2
32514	5.97	1	0.84	0.64	2.16	2.05	1.8	1.93
32541	6.06	1	0.97	0.67	1.79	2	1.81	2.04
34125	6.24	1	1.12	0.63	0.56	0.41	0.14	1.88
34152	6.56	1	1.05	0.77	0.64	1.28	1.26	2.09
34215	4.76	1	0.82	0.78	0.48	0.41	0.23	1.59
34251	5.22	1	0.99	0.84	0.46	1.74	1.58	1.76
34512	6.45	1	1.05	0.84	2.14	2.16	1.88	2.1
34521	6.01	1	0.83	0.76	2.09	1.79	1.85	2.02
35124	8.43	1	1.05	2.39	2.6	2.36	2.6	2.68
35142	9.34	1	1.2	3.02	2.78	3.43	2.97	2.99
35214	8.04	1	1.04	2.37	2.48	2.48	2.47	2.93
35241	8.5	1	1.08	2.68	2.75	2.55	2.68	2.62
35412	8.66	1	1	2.44	2.77	2.75	2.67	2.86
35421	9.11	1	1.04	2.61	3	2.78	2.86	3.03

**Figure 40: lbnl-network results normalized by splatt (parallel) (4)**

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	4-sadilla	k-sadilla	radix
41235	6.61	1	1.24	0.62	0.4	0.34	$7.7 \cdot 10^{-2}$	2.1
41253	6.2	1	1.04	0.53	0.33	1.25	1.24	1.98
41325	4.17	1	0.99	0.57	0.46	0.4	0.26	1.27
41352	4.57	1	0.99	0.62	0.56	1.14	1.18	1.5
41523	5.02	1	1.03	0.54	1.07	1.06	1.02	1.71
41532	5.12	1	1	0.55	1.09	1.24	1.28	1.67
42135	3.67	1	1.13	0.21	0.2	0.18	$8 \cdot 10^{-2}$	1.09
42153	3.88	1	0.97	0.21	0.17	0.73	0.73	1.21
42315	3.06	1	0.84	0.21	0.26	0.2	0.18	0.98
42351	3.99	1	1	0.27	0.27	1.23	1.21	1.28
42513	3.68	1	0.89	0.19	1.2	1.27	1.14	1.17
42531	3.74	1	0.82	0.2	1.2	1.17	1.22	1.15
43125	3.28	1	1.07	0.56	0.31	0.24	$7.1 \cdot 10^{-2}$	1.01
43152	2.87	1	0.86	0.5	0.3	0.61	0.55	0.92
43215	3.17	1	0.98	0.75	0.3	0.28	0.14	1.03
43251	3.88	1	1.04	0.81	0.34	1.21	1.23	1.15
43512	3.64	1	1.24	0.71	1.07	1.24	1.11	1.18
43521	3.63	1	1.24	0.63	1.17	1.07	1.04	1.18
45123	4.47	1	0.88	1.23	1.23	1.22	1.19	1.34
45132	4.28	1	1.13	1.22	1.22	1.51	1.48	1.36
45213	4.37	1	1.18	1.27	1.38	1.41	1.3	1.49
45231	4.72	1	1.1	1.3	1.37	1.53	1.39	1.57
45312	4.4	1	0.98	1.31	1.32	1.37	1.25	1.32
45321	4.27	1	0.89	1.29	1.39	1.28	1.39	1.36

**Figure 41: lbnl-network results normalized by splatt (parallel) (5)**

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	4-sadilla	k-sadilla	radix
51234	1.88	1	0.96	0.53	0.5	0.5	0.57	0.61
51243	1.86	1	1.02	0.51	0.5	0.6	0.61	0.58
51324	1.86	1	1	0.5	0.61	0.61	0.58	0.57
51342	1.86	1	1.03	0.49	0.62	0.68	0.66	0.6
51423	1.89	1	1.01	0.5	0.59	0.6	0.58	0.59
51432	1.93	1	1.04	0.56	0.61	0.69	0.69	0.59
52134	1.95	1	1.04	0.57	0.55	0.56	0.53	0.61
52143	1.88	1	1	0.55	0.55	0.6	0.57	0.57
52314	1.91	1	1.05	0.54	0.57	0.57	0.56	0.64
52341	1.85	1	1.02	0.54	0.55	0.59	0.56	0.6
52413	1.87	1	0.98	0.54	0.55	0.55	0.57	0.58
52431	1.88	1	0.98	0.55	0.66	0.58	0.59	0.58
53124	1.92	1	1.06	0.55	0.54	0.53	0.53	0.62
53142	1.92	1	1.04	0.55	0.55	0.66	0.61	0.63
53214	1.87	1	0.99	0.56	0.56	0.58	0.59	0.62
53241	1.88	1	1.01	0.54	0.57	0.58	0.56	0.56
53412	1.93	1	1.04	0.55	0.56	0.56	0.6	0.61
53421	1.79	1	0.99	0.51	0.55	0.54	0.55	0.58
54123	1.94	1	1.01	0.57	0.55	0.53	0.57	0.62
54132	1.91	1	1.06	0.52	0.54	0.62	0.6	0.64
54213	1.87	1	0.98	0.53	0.63	0.56	0.55	0.58
54231	1.87	1	0.99	0.55	0.61	0.61	0.59	0.58
54312	1.92	1	1.04	0.52	0.55	0.57	0.59	0.59
54321	1.86	1	1.03	0.53	0.58	0.6	0.57	0.56

**Figure 42: vast-2015-mc1-5d results normalized by splatt (parallel) (1)**

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	4-sadilla	k-sadilla	radix
12345	19.99	1	0.29	0.27	0.27	0.28	0	6.29
12354	19.86	1	0.28	0.28	0.27	2.17	1.9	6.24
12435	19.58	1	0.28	0.27	2.14	2.17	1.9	6.14
12453	19.43	1	0.28	0.26	2.17	4.14	3.75	6.3
12534	19.5	1	0.29	0.27	2.16	2.2	1.86	6.25
12543	19.49	1	0.28	0.26	2.14	4.04	3.76	6.07
13245	19.23	1	0.33	1.41	1.52	1.52	1.22	5.92
13254	19.13	1	0.33	1.47	1.54	3.4	3.06	5.76
13425	21.28	1	0.44	1.31	2.36	2.43	2.18	4.41
13452	21.23	1	0.45	1.27	2.32	3.61	3.34	4.29
13524	22.13	1	0.44	1.3	2.42	2.43	2.21	4.38
13542	21.97	1	0.45	1.28	2.29	3.57	3.33	4.4
14235	21.99	1	0.41	1.42	1.47	1.51	1.23	4.65
14253	23.11	1	0.41	1.46	1.48	3.06	2.8	4.73
14325	22.74	1	0.42	1.44	2.49	2.57	2.3	4.68
14352	22.65	1	0.42	1.41	2.43	3.75	3.57	4.54
14523	22.3	1	0.42	1.4	2.71	2.65	2.48	4.62
14532	22.42	1	0.42	1.42	2.72	3.69	3.58	4.6
15234	23.33	1	0.41	1.45	1.49	1.51	1.24	4.74
15243	23.33	1	0.41	1.44	1.49	3.1	2.79	4.9
15324	22.05	1	0.42	1.43	2.47	2.49	2.33	4.68
15342	22.87	1	0.43	1.48	2.51	3.83	3.55	4.61
15423	23.21	1	0.43	1.48	2.84	2.84	2.48	4.68
15432	22.6	1	0.42	1.41	2.67	3.77	3.5	4.58

Figure 43: vast-2015-mc1-5d results normalized by splatt (parallel) (2)

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	4-sadilla	k-sadilla	radix
21345	27.03	1	1.17	1.22	1.2	1.24	1.01	4.27
21354	27.35	1	1.17	1.24	1.21	2.89	2.66	4.2
21435	27.9	1	1.2	1.23	2.89	2.94	2.85	4.17
21453	26.41	1	1.16	1.22	2.9	4.49	4.3	4.08
21534	26.3	1	1.16	1.26	2.95	2.92	2.67	4.1
21543	28.65	1	1.21	1.3	2.96	4.71	4.53	4.32
23145	25.02	1	1.16	1.64	1.72	1.67	1.61	3.7
23154	24.95	1	1.16	1.66	1.67	3.11	2.97	3.73
23415	23.57	1	1.15	1.69	1.94	1.97	1.8	3.39
23451	23.8	1	1.12	1.69	1.97	2.43	2.24	2.74
23514	22.63	1	1.12	1.66	1.92	1.94	1.78	3.22
23541	23.32	1	1.14	1.66	1.93	2.45	2.28	2.71
24135	23.25	1	1.13	1.52	1.56	1.59	1.37	3.39
24153	24.96	1	1.15	1.56	1.58	2.95	2.8	3.65
24315	23.72	1	1.15	1.57	2.05	2.07	1.9	3.49
24351	24.66	1	1.16	1.66	2.11	2.64	2.4	2.95
24513	23.57	1	1.14	1.63	2.08	2.05	1.9	2.91
24531	23.5	1	1.14	1.59	2.09	2.52	2.42	2.89
25134	23.42	1	1.14	1.55	1.58	1.56	1.37	3.5
25143	24.98	1	1.14	1.54	1.57	3.01	2.8	3.49
25314	22.98	1	1.15	1.55	2.06	2.08	1.91	3.52
25341	23.52	1	1.14	1.65	2.08	2.6	2.37	2.83
25413	24.15	1	1.18	1.65	2.15	2.15	1.96	3.02
25431	24.38	1	1.13	1.64	2.11	2.62	2.45	2.92

Figure 44: vast-2015-mc1-5d results normalized by splatt (parallel) (3)

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	4-sadilla	k-sadilla	radix
31245	2.97	1	1.07	0.15	0.16	0.16	0.12	0.9
31254	3.01	1	1.06	0.15	0.16	0.43	0.39	0.89
31425	1.84	1	0.99	$7.4 \cdot 10^{-2}$	0.17	0.17	0.15	0.38
31452	1.77	1	0.99	$7.2 \cdot 10^{-2}$	0.16	0.26	0.25	0.34
31524	1.88	1	1	$7.5 \cdot 10^{-2}$	0.17	0.17	0.15	0.37
31542	1.79	1	1	$7.4 \cdot 10^{-2}$	0.16	0.27	0.24	0.35
32145	1.35	1	1	$9.3 \cdot 10^{-2}$	$9.3 \cdot 10^{-2}$	$9.2 \cdot 10^{-2}$	$8.2 \cdot 10^{-2}$	0.21
32154	1.41	1	1	$9.4 \cdot 10^{-2}$	$9.3 \cdot 10^{-2}$	0.17	0.16	0.21
32415	1.31	1	0.99	$9.7 \cdot 10^{-2}$	0.11	0.12	0.11	0.2
32451	1.36	1	1	$9.8 \cdot 10^{-2}$	0.11	0.14	0.13	0.15
32514	1.32	1	1	$9.9 \cdot 10^{-2}$	0.12	0.12	0.11	0.19
32541	1.32	1	1	$9.5 \cdot 10^{-2}$	0.11	0.14	0.13	0.16
34125	1.42	1	0.99	$6.7 \cdot 10^{-2}$	$6.4 \cdot 10^{-2}$	$6.4 \cdot 10^{-2}$	$5.2 \cdot 10^{-2}$	0.2
34152	1.37	1	0.99	$7.1 \cdot 10^{-2}$	$6 \cdot 10^{-2}$	0.12	0.11	0.17
34215	1.43	1	1	$9.2 \cdot 10^{-2}$	0.1	0.1	$9.3 \cdot 10^{-2}$	0.18
34251	1.43	1	0.99	$8.9 \cdot 10^{-2}$	0.1	0.12	0.11	0.14
34512	1.28	1	0.99	$9.2 \cdot 10^{-2}$	$7.7 \cdot 10^{-2}$	$8 \cdot 10^{-2}$	$7 \cdot 10^{-2}$	0.17
34521	1.45	1	1	$9.5 \cdot 10^{-2}$	$8.8 \cdot 10^{-2}$	0.12	0.11	0.14
35124	1.41	1	0.99	$6.3 \cdot 10^{-2}$	$6.2 \cdot 10^{-2}$	$6.3 \cdot 10^{-2}$	$5.2 \cdot 10^{-2}$	0.19
35142	1.33	1	1	$6.6 \cdot 10^{-2}$	$6.2 \cdot 10^{-2}$	0.11	0.11	0.18
35214	1.44	1	1	$8.6 \cdot 10^{-2}$	0.1	0.1	$9.4 \cdot 10^{-2}$	0.18
35241	1.45	1	1	$8.2 \cdot 10^{-2}$	0.1	0.12	0.11	0.14
35412	1.31	1	0.99	$8.2 \cdot 10^{-2}$	$7.9 \cdot 10^{-2}$	$7.8 \cdot 10^{-2}$	$7.1 \cdot 10^{-2}$	0.17
35421	1.41	1	1	$8.3 \cdot 10^{-2}$	$8.9 \cdot 10^{-2}$	0.12	0.11	0.13

**Figure 45: vast-2015-mc1-5d results normalized by splatt (parallel) (4)**

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	4-sadilla	k-sadilla	radix
41235	41.46	1	1.12	1.06	1.06	1.06	0.79	5.83
41253	41.01	1	1.13	1.08	1.08	3.04	2.71	5.98
41325	40.9	1	1.14	1.03	2.4	2.54	2.16	5.8
41352	36.31	1	1.12	0.95	2.19	3.78	3.46	4.76
41523	36.89	1	1.11	0.94	2.59	2.54	2.33	4.85
41532	36.84	1	1.12	0.96	2.55	3.76	3.47	4.83
42135	25.27	1	1.05	1.36	1.32	1.33	1.19	3.05
42153	24.67	1	1.06	1.34	1.39	2.6	2.42	3.13
42315	25.02	1	1.08	1.39	1.79	1.79	1.65	3.16
42351	24.31	1	1.08	1.48	1.85	2.18	2.04	2.5
42513	25.46	1	1.07	1.46	1.85	1.86	1.63	2.64
42531	24.09	1	1.05	1.42	1.72	2.21	2.04	2.51
43125	30.43	1	1.06	1.5	1.34	1.35	1.15	4.39
43152	28.6	1	1.06	1.46	1.29	2.57	2.34	3.74
43215	22.53	1	1.06	1.47	1.61	1.63	1.48	2.84
43251	23.44	1	1.05	1.47	1.67	2.03	1.84	2.25
43512	21.04	1	1.04	1.49	1.26	1.26	1.09	2.78
43521	23.26	1	1.07	1.53	1.46	1.91	1.8	2.22
45123	22.31	1	1.06	0.93	0.97	0.97	0.83	3.06
45132	22.91	1	1.05	0.96	0.99	1.72	1.54	3.04
45213	24.26	1	1.07	1.12	1.63	1.63	1.54	2.35
45231	23.4	1	1.05	1.12	1.63	2.07	1.92	2.29
45312	21.81	1	1.04	0.98	1.32	1.34	1.18	2.91
45321	22.46	1	1.06	1.15	1.44	1.94	1.8	2.22

**Figure 46: vast-2015-mc1-5d results normalized by splatt (parallel) (5)**

	qsort	splatt	1-sadilla	2-sadilla	3-sadilla	4-sadilla	k-sadilla	radix
51234	42.16	1	1.17	1.07	1.07	1.07	0.78	5.95
51243	41.33	1	1.15	1.08	1.07	3	2.65	6
51324	41.2	1	1.13	1.05	2.41	2.41	2.12	5.87
51342	37.04	1	1.1	1	2.19	3.8	3.57	4.84
51423	37.65	1	1.09	0.99	2.59	2.63	2.31	4.84
51432	38.03	1	1.14	1.01	2.65	4	3.59	4.97
52134	25.57	1	1.07	1.33	1.35	1.35	1.2	3.06
52143	24.29	1	1.06	1.32	1.34	2.52	2.47	3.01
52314	24.48	1	1.05	1.32	1.75	1.72	1.64	3.01
52341	24.32	1	1.06	1.43	1.8	2.2	2.09	2.5
52413	25.35	1	1.08	1.42	1.8	1.81	1.65	2.58
52431	24.51	1	1.06	1.42	1.79	2.21	2.09	2.48
53124	30.37	1	1.05	1.46	1.34	1.38	1.13	4.28
53142	28.81	1	1.06	1.46	1.31	2.61	2.36	3.74
53214	23.36	1	1.05	1.49	1.62	1.65	1.51	2.83
53241	23.36	1	1.04	1.45	1.63	1.97	1.87	2.26
53412	21.75	1	1.06	1.55	1.33	1.31	1.15	2.78
53421	22.62	1	1.05	1.46	1.38	1.84	1.7	2.1
54123	23	1	1.06	0.95	0.99	0.98	0.83	3.07
54132	22.55	1	1.06	0.95	0.98	1.7	1.54	2.97
54213	23.99	1	1.06	1.09	1.65	1.61	1.47	2.29
54231	23.14	1	1.06	1.13	1.66	2.02	1.86	2.3
54312	22.17	1	1.05	0.96	1.34	1.36	1.17	2.94
54321	23.06	1	1.05	1.13	1.49	1.97	1.84	2.22