Toward Parallel Document Clustering

Jace A. Mogill, David J. Haglin

Pacific Northwest National Laboratory
Richland, WA, 99354 USA
{jace.mogill, david.haglin}@pnl.gov

***Abstract*—A key challenge to automated clustering of documents in large text corpora is the high cost of comparing documents in a multi-million dimensional document space. The Anchors Hierarchy is a fast data structure and algorithm for localizing data based on a triangle inequality obeying distance metric, the algorithm strives to minimize the number of distance calculations needed to cluster the documents into "anchors" around reference documents called "pivots". We extend the original algorithm to increase the amount of available parallelism and consider two implementations: a complex data structure which affords efficient searching, and a simple data structure which requires repeated sorting. The sorting implementation is integrated with a text corpora "Bag of Words" program and initial performance results of end-to-end document processing workflow are reported.**

I. Introduction

Automated clustering of documents in large text corpora begins with the construction of document signatures in the form of sparse vectors of word frequencies. This phase is a "Bag of Words" (BoW) calculation, which is described in section II-A. The BoW processing is followed by anchors hierarchy's clustering of points into "anchors", which are sets of points ordered by distance from a "pivot". The notion is similar to a cluster with a centroid but is relaxed in the sense a pivot is not the geometric center of a cluster, rather it is one of the data points serving as the point of reference for the other points in the anchor, and for that anchor relative to other anchors. The data structure explicitly stores inter-pivot distances in sorted lists to reduce recalculation and accelerate searching for points.

The serial clustering process, seen in figure 1, is not suitable for use on parallel computers due to decreasing amounts of concurrency as clustering proceeds. Parallelizing the serial algorithm requires restructuring loops and data structures to break data dependencies which would normally be enforced through the serial outer loop. This technique is presented in II-C.

We conclude with early results for a multi-threaded implementation of the anchors hierarchy applied to large text corpus, and identify several areas of further research both in algorithmic efficacy and execution performance on both the Cray XMT and ordinary multi-core x86 systems.

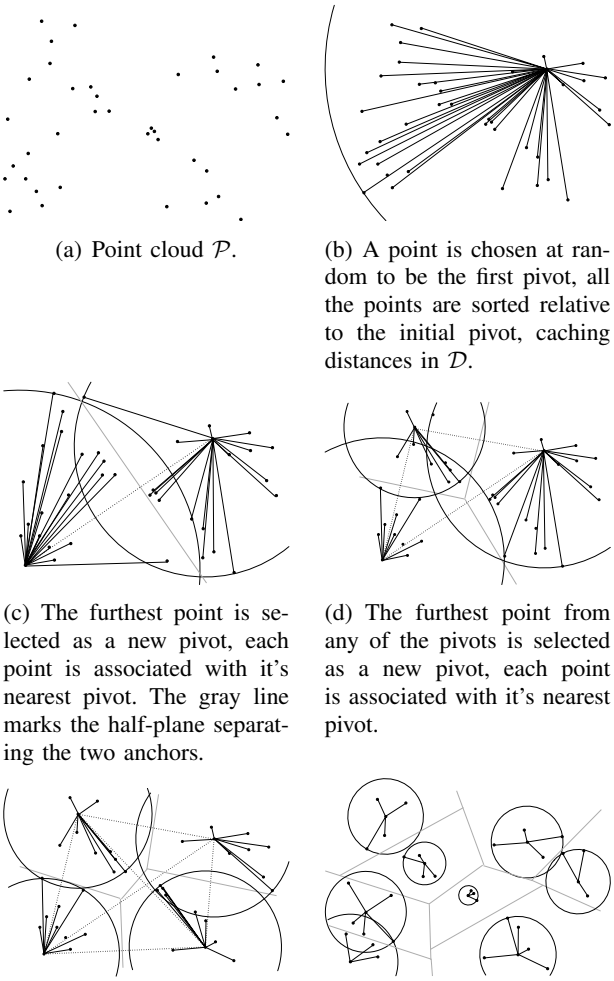
II. Algorithms

The two major components of our application are the parsing and indexing of the text corpus, and the clustering of the documents. The text processing is dominated by a *Bag of Words* operation, described in section II-A, and deals with tokenization of strings, mapping unique words (tokens) to integers, and counting the frequency of word occurrence per document. The Anchors Hierarchy phase, described in section II-B, performs the clustering of the documents into a navigable structure.

A. Bag of Words

Given a document of words—such as a web page with text on it, an email, or a book—it is possible to characterize the document by counting the frequency of each word it contains. Note that this characterization ignores the order of the words within the document. The collection of word-to-frequency mappings is called a *Bag of Words* (BoW).

One of the most important applications of BoW is to provide *document signatures*—a strategy for characterizing documents within a corpus—and to use these signatures as a way to measure similarity between documents. The similarity measure can



(a) Point cloud $\mathcal{P}$.

(b) A point is chosen at random to be the first pivot, all the points are sorted relative to the initial pivot, caching distances in $\mathcal{D}$.

(c) The furthest point is selected as a new pivot, each point is associated with it's nearest pivot. The gray line marks the half-plane separating the two anchors.

(d) The furthest point from any of the pivots is selected as a new pivot, each point is associated with it's nearest pivot.

(e) Step d is repeated until the desired number of anchors have been created.

(f) Completed anchor formation. The half-plane lines form a Voronoi diagram of the anchor space.

Fig. 1: Serial anchor formation. Anchors are added one at a time until the desired number have been formed.

then be used to search for clusters [1] of similar documents.

The signature is a sequence of (*wordID*, *frequency*) pairs. This information can be interpreted as a sparse vector whose size is the number of unique words in the entire corpus, each *wordID* represents a position within the vector, and the *frequency* represents the magnitude in that dimension. Thus, the BoW places documents into $\mathcal{R}^n$ space, where n is the number of unique words found in the entire corpus, and any distance measure in the n-dimensional space can be used as a similarity measure. This vector depiction is called a *sparse vector* since most documents contain a small subset of the n words in the lexicon, which means the magnitude of most dimensions is zero. Considering a document signature as a sparse vector is a well-established approach to characterizing documents [2].

In statistical text modeling, a common assumption used by models like Latent Dirichlet Allocation [3] is that of *exchangeability* of words within a document [4], which requires that the probability of a given sequence of words is independent of the order those words appear. Under this assumption, a given corpus can be faithfully represented by a BoW representation.

The fundamentals of the BoW problem induce algorithmic solutions that will exhibit both *regular* and *irregular* memory access patterns. The reasons are that the input corpus is typically read in a sequential manner with very high spatial locality, while the assignment of unique identifiers to words is inherently irregular, since this assignment can be efficiently done using a hash-based mapping of strings to integers. Given the unpredictable appearances of words in the input corpus the accesses to such mapping will be highly irregular.

B. Anchors Hierarchy

The Anchors Hierarchy [5] gets around problems of high-dimensionality by ignoring the concrete data representation enitirely, and instead leveraging the properties of a general metric space. For any points x, y, and z, we require:

$$\begin{aligned} d(x,y) &= 0 \text{ iff } x = y \\ d(x,y) &= d(y,x) \\ d(x,z) &\leq d(x,y) + d(y,z) \end{aligned}$$

The last of the above is called the *triangle inequality*. The primary cost savings in the anchors hierarchy comes from reducing the number of distance calculations by paying careful attention to the implications of the triangle inequality.

For the experiments in this paper, we define the distance between two document vectors x and y using the cosine distance

$$d(x,y) = \arccos\left(\frac{x \cdot y}{\|x\| \, \|y\|}\right).$$

If x and y are normalized to have unit length, this simplifies to $d(x, y) = \arccos(x \cdot y)$. The document vectors are normalized to intersect the unit sphere, and the distance between two documents is the great circle distance on the sphere. Maximal distance occurs when documents have no words in common, in which case the dot product is zero, leading to a distance of $\pi/2$.

Rather than implement cosine distance from normalized words frequencies directly as we have, it is more common to use an approach like tf-idf [6] that takes into account the occurrence of a term across documents. We do not anticipate any difficulties in parallelizing tf-idf normalization, and expect to implement it in a later phase of the project. We look forward to future work experimenting with a variety of different distance metrics.

Given a large text corpus, we first build the bag of words representation, as described in [7]. The resulting sparse vectors are then passed to build a set of anchors, which are then merged hierarchically as described in section II-D. Following this, a typical analysis would feed to any number of summarization algorithms or query tools; we do not yet address these.

1) Constants and Notation: The two important constants in the parallel algorithm are K, the total number of anchors to create, and e, the number of new anchors to add simultaneously. It is not necessary for the value of e to be constant – the value may be recomputed before new anchors are chosen. The following notation is used:

- N is the number of points
- n is the number of dimensions
- K is the total number of anchors to create
- k is the number of anchors presently
- $\mathcal{P}$ is the set of all points, where:
$$\mathcal{P} = \{p_1, ..., p_N\}$$
- $\mathcal{A}$ is the set of pivots, $\mathcal{A} = \{a_1, ..., a_K\}$
- $\mathcal{S}$ is the set of anchors, $\mathcal{S} = \{\mathcal{C}_1, ..., \mathcal{C}_K\}$
- $\mathcal{C}_i$ is an anchor of points:
$$\mathcal{C}_{i \in \{1..k\}} = \mathcal{P}_{idx(i, j \in \{1..|\mathcal{C}_i|\})}$$
- $\mathcal{D}$ is the set of distances from points to their anchors, $\mathcal{D}_{i \in \{1..k\}} = \|\mathcal{A}_i - \mathcal{P}_{idx(i, j \in \{1..|\mathcal{C}_i|\})}\|$

The function $idx(i, j)$ converts an anchor name i and rank index j of a point in the anchor $\mathcal{C}_i$ to the index of the point in $\mathcal{P}$, the set of all points. A point is in exactly one anchor, such that $P = \mathcal{C}_{idx(i \in \{1..k\}, j \in \{1..|\mathcal{C}_i|\})}$.

The conventional set notation is used throughout the examples carrying the additional semantics that elements in the set are ordered and are individually accessible by indexes. Specifically, point $p_{idx(i,j)}$ is the j^{th} most distant point in anchor i from the pivot, $\mathcal{A}_i$. Furthermore, when set union and subtraction are performed, we imply the ordering is preserved. For example, $\mathcal{C}_i \backslash \mathcal{P}_{idx(i,m)}$ indicates the m^{th} point in anchor i is removed from that anchor, implying that points $\mathcal{P}_{m+1..|\mathcal{C}_i|}$ are renamed $\mathcal{P}_{m..|\mathcal{C}_i - 1|}$. Similarly, $\mathcal{C}' \cup \mathcal{P}_{idx(i,m)}$ inserts the point $\mathcal{P}_{idx(i,m)}$ into the set $\mathcal{C}'$ at the index position corresponding to it's rank in the distance ordered list from the anchor's pivot, $\mathcal{A}_i$, and the points $\mathcal{C}'_{\{m..k\}}$ are renamed $\mathcal{C}'_{\{m+1..k+1\}}$.

2) Serial Algorithm: The serial Anchors Hierarchy algorithm, shown in figure 2 proceeds as follows: Initialization begins by selecting a first pivot at random from the set of points and assigning the remaining points to the anchor of the initial pivot (lines 1-2). The distance from each point to the pivot is calculated and cached in $\mathcal{D}$ (line 3). The outer loop has $K - 1$ iterations, each iteration will turn exactly one point from an ordinary anchor point into a pivot point for a new anchor.

Every iteration begins with selection of a new pivot point, a', which is the point furthest from any existing pivot. It is then removed from the anchor it comes from (lines 5-7). The set of points associated with the new anchor, $\mathcal{C}'$, is initialized as empty (line 8).

The new pivot is compared to every existing pivot, if the distance from the new pivot to the pivot under consideration ($\|a' - a_i\|$) is more than twice the radius of that anchor ($\|\mathcal{C}_{i_m} - a_i\|$), no points in the anchor under consideration can be nearer to the new pivot than it's current pivot and the entire anchor may be skipped (lines 10-12). If it is possible for points in the anchor under consideration to be closer to the new pivot, the points in the anchor are considered one at a time, in order of descending distance to the old pivot (lines 13-18). Points which are closer to the new pivot than the old pivot are moved from the old anchor, $\mathcal{C}_i$, to the new anchor, $\mathcal{C}'$, on lines 13-18. When the distance of the point being considered to it's old pivot is less than half the distance from the new

Procedure: Serial Anchors Hierarchy Algorithm
Input: Point cloud $\mathcal{P} = \{p_1, ..., p_N\}$
Input: K = Number of anchors to form
Output: Anchors of points $\mathcal{C} = \mathcal{P}_{idx(i \in \{1..k\}, j \in \{1..|\mathcal{C}_i|\})}$
Output: Set of pivots $\mathcal{A} = \{a_1, ..., a_K\}$

1: $a_1 \leftarrow$ Anchor point, selected randomly from $\mathcal{P}$
2: $\mathcal{C}_1 \leftarrow \mathcal{P} \setminus a_1$
3: $\mathcal{D}_{i \in \{1..k\}} = \|\mathcal{A}_i - \mathcal{P}_{idx(i, j \in \{1..|\mathcal{C}_i|\})}\|$
4: **for all** k in $\{1..K-1\}$ **do**
5: $\quad u \leftarrow \arg\max_{u \in \{1..k\}} \mathcal{D}_{idx(u,|\mathcal{C}_u|)}$
6: $\quad a' \leftarrow \mathcal{P}_{idx(u,|\mathcal{C}_u|)}$
7: $\quad \mathcal{C}_u \leftarrow \mathcal{C}_u \setminus a'$
8: $\quad \mathcal{C}' \leftarrow \emptyset$
9: $\quad$ **for all** i in $\{1..k\}$ where $i \neq u$ **do**
10: $\quad\quad h \leftarrow \frac{1}{2}\|a' - a_i\|$
11: $\quad\quad m \leftarrow |C_i|$
12: $\quad\quad$ **while** $m \geq 0$ and $\mathcal{D}_{idx(i,m)} > h$ **do**
13: $\quad\quad\quad d \leftarrow \|a' - \mathcal{P}_{idx(i,m)}\|$
14: $\quad\quad\quad$ **if** $d < \mathcal{D}_{idx(i,m)}$ **then**
15: $\quad\quad\quad\quad \mathcal{C}' \leftarrow \mathcal{C}' \cup \mathcal{P}_{idx(i,m)}$
16: $\quad\quad\quad\quad \mathcal{C}_i \leftarrow \mathcal{C}_i \setminus \mathcal{P}_{idx(i,m)}$
17: $\quad\quad\quad\quad \mathcal{D}' \leftarrow \mathcal{D}' \cup d$
18: $\quad\quad\quad\quad \mathcal{D}_i \leftarrow \mathcal{D}_i \setminus d$
19: $\quad\quad\quad$ **end if**
20: $\quad\quad\quad m \leftarrow m - 1$
21: $\quad\quad$ **end while**
22: $\quad$ **end for**
23: $\quad \mathcal{A} \leftarrow \mathcal{A} \cup a'$
24: $\quad \mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{C}'$
25: $\quad \mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}'$
26: **end for**

Fig. 2: Serial Anchors Hierarchy Algorithm.

pivot to the old pivot the search may stop. After all the existing anchors have been considered, the new pivot and it's associated points are committed to the sets of completed anchors (lines 23-25).

C. Parallel Algorithm

The Parallel Anchors Hierarchy algorithm extends the serial algorithm by adding multiple anchors simultaneously. This is made possible by breaking the dependence carried by the outer loop through allocating temporary storage to perform a reduction incrementally, the reduction can then be moved inside it's own dedicated parallel loop.

1) Initialization: The initialization of the parallel algorithm randomly selects the initial pivot and computes the distance from every other point to the pivot to form the first anchor.

2) New Anchor Selection: Instead of associating points to the new anchor before selecting the next new anchor as in the serial algorithm, the parallel algorithm selects e anchors simultaneously. This process can be deterministic, but does not produce the same results as the serial algorithm. Because the results will not match the serial results, implementers are free to use any point selection method, for example: furthest e points from any pivot(s), furthest point from each of e pivots, random, etc.

The examples in this paper uses the furthest of e pivots method. The point is moved to the set of new anchors, $\mathcal{A}'$, and removed from the cluster it comes from, $\mathcal{C}_i$.

The number of new anchors to introduce at one time, e, may be any value $1..K$, Moore suggests $\sqrt{N}$ in [5]. A value of 1 introduces only one new anchor at a time, and exactly replicates the semantics of the original serial algorithm. At the other extreme, a value of K introduces all the new anchors at once, which is equivalent to a brute force search.

3) Associating Points with New Anchors: In contrast to the serial algorithm which moves a point to a new anchor as soon as the point is inspected, the parallel algorithm (figure 3) must first consider moving each point to every new anchor before the point is finally moved to the pivot which it is nearest. This operation can be thought of as an unstructured reduction, implemented by splitting the reduction process into two steps: one parallel region to decide if/where the point should move (distance minimization), and another to actually move the point.

The same distance calculations (lines 6-9) as performed in the serial algorithm (figure 2, lines 9-14) are used to eliminate anchors more than $\frac{1}{2}(\|a'_j - a_i\| - \|a_i - \mathcal{D}_{i_m}\|)$ away from the new pivot being considered. The new candidate anchor's name (j) is stored in $\mathbb{X}_{i_m}$ (line 10), indicating to which anchor this point will move at the end of the phase.

In order to permit unordered updates to new

Procedure: Associate Points with New Anchors
Input: Anchor of points $\mathbb{C}$
Input: Set of new pivots $\mathcal{A}'$
Input: Number of clusters formed so far, k
Input: Number of new anchors e
Output: Future Anchor associations $\mathbb{X}$

1: $\mathcal{X} \leftarrow$ Inverse index of $\mathbb{C}$
2: $\mathcal{D}_{i \in \{1..k\}} = \|\mathcal{A}_i - \mathcal{P}_{idx(i,j \in \{1..|\mathcal{C}_i|\})}\|$
3: **for all** i in $\{1..k\}$ **do** ▷ Parallel Loop
4: $m \leftarrow \underset{m \in \{1..|\mathcal{D}_i|\}}{\arg\max} (\|a_i - \mathcal{D}_{i_m}\|)$
5: **for all** j in $\{1..e\}$ **do** ▷ Parallel Loop
6: $h \leftarrow \frac{1}{2}\|a'_j - a_i\|$
7: **while** $\|a'_j - \mathcal{D}_{i_m}\| < h$ **do**
8: $t \leftarrow \mathcal{X}_{i_m}$
9: **if** $(\|a'_j - \mathcal{D}_{i_m}\| < \|a_t - \mathcal{D}_{i_m}\|)$ **then**
10: $\mathcal{X}_{i_m} \leftarrow j$
11: **end if**
12: $D_i \leftarrow D_i \setminus D_{i_m}$
13: $m \leftarrow \underset{m \in \{1..|\mathcal{D}_i|\}}{\arg\max} (\|a_i - \mathcal{D}_{i_m}\|)$
14: **end while**
15: **end for**
16: **end for**

Fig. 3: Parallel method for associating points with new anchors.

whereas the parallel algorithm uses the storage of $\mathbb{X}$ to schedule their moves in a separate phase. Figure 4 shows how this is performed.

Procedure: Move Points to New Anchors
Input: Anchors $\mathbb{C}$
Input: Set of new pivots $\mathcal{A}'$
Input: Number of clusters formed so far, k
Input: Future Anchor associations $\mathbb{X}$
Output: Anchors $\mathbb{C}$
Output: Set of Pivots $\mathcal{A}$
Output: Number of anchors formed at end of phase, k

1: $\mathbb{C}' \leftarrow \emptyset$
2: **for all** i in $\mathbb{C}$, j in $\mathcal{C}_i$ **do** ▷ Parallel Loop
3: $t \leftarrow \mathcal{X}_{i_j}$
4: **if** $\mathcal{X}_{i_j} \neq i$ **then**
5: $\mathcal{C}'_t \leftarrow \mathcal{C}'_t \cup \mathcal{C}_{i_j}$
6: $\mathcal{C}_i \leftarrow \mathcal{C}_i \setminus \mathcal{C}_{i_j}$
7: **end if**
8: **end for**
9: $\mathbb{C} \leftarrow \mathbb{C} \cup \mathbb{C}'$
10: $\mathcal{A} \leftarrow \mathcal{A} \cup \mathcal{A}'$
11: $k \leftarrow k + e$

Fig. 4: Parallel method for moving points from old anchors to new anchors and committing new anchors to the new anchors list.

anchors stored in $\mathbb{X}$, the updates must be made atomic with respect to to other iterations as well as comparisons. Machines such as the Cray XMT in which synchronization is abundant [8] allow a specific point (i.e., $\mathbb{X}_{i_m}$) to be "emptied" preventing other threads from using that point until it is "filled", thereby enforcing atomic update semantics. On systems where synchronization must be rationed, the N possible update sites could be mapped onto a smaller number of locks. Alternatively, the threads themselves may be synchronized by turning lines 8-12 into a critical region in which only one thread can execute at a time. Fine grained synchronization affords N degrees of concurrency, lock coloring reduces the concurrency to L locks, and critical regions have no concurrency at all. The three techniques produce identical output.

4) Move Points to New Anchors: The serial algorithm moves points to new anchors incrementally,

The point motion loop is embarrassingly parallel in that every point may be moved simultaneously. The loop on line 2 iterates through every anchor and every point in every anchor comparing the new pivot stored in $\mathbb{X}$ to the current pivot. If the point is to be moved, it is inserted into the new anchor and removed from the old anchor (lines 4-6). After all points have been processed, the set of new pivots is merged with the old set of pivots, and the set of new anchors is merged with the old set of anchors (lines 9-10). Finally, the number of anchors found so far, k, is updated to the current number of anchors.

D. Form Hierarchy of Anchors

The second phase of the Anchors Hierarchy, following the clustering of points into anchors, is hierarchy formation where the anchors are combined into a navigable structure. The hierarchy formation process recursively merges the two anchors which

when combined form the smallest new possible anchor. The two anchors are then replaced with their combined anchor, the process is repeated, the final hierarchy is illustrated in figure 5.

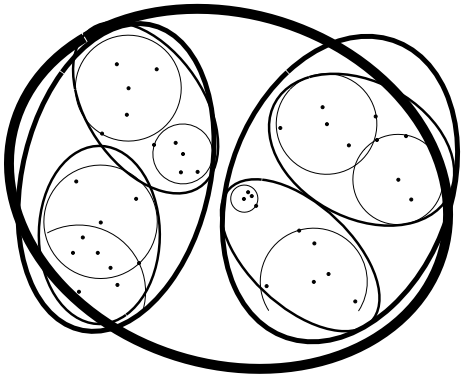


Fig. 5: The complete Anchors Hierarchy after multiple rounds of anchor combining.

The hierarchy of anchors can be thought of as a pennant tree, a special form of a binary tree in which a parent node has one unique child and itself for the other child [9]. Instead of combining K anchors in $K-1$ phases, pennant tree construction can proceed in parallel by opportunistically combining all pairs of anchors which mutually form the smallest new anchor. Anchors which do not mutually pair during one round of combining are propagated to the next round with the newly combined anchors (figure 6).

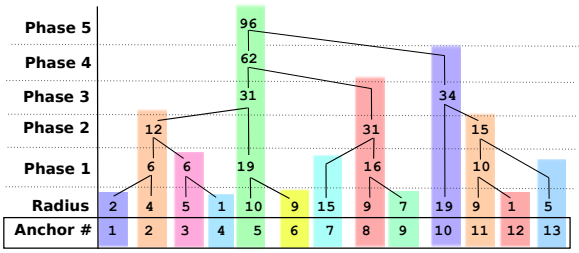


Fig. 6: Parallel formation of a pennant tree by creating the smallest new anchor from two extant anchors.

III. Implementation

Direct implementation of the parallel Anchors Hierarchy algorithm presents many challenges for efficient execution because the parallelism effectively inverts from the outermost loop to innermost over the course of execution, requiring efficient load balancing. Worse, serial inner loops may impose nearly pathological load imbalances. Relative to x86, the serial performance penalty of the Cray XMT can be on the order of 100x, and unbalanced load readily defeats the parallelism of a 100 processor system.

Ahmdal's Law requires us to not only parallelize but also load balance 99.999% of the program, effectively eliminating algorithms and data structures with an implied serial inner loop (i.e.: the linked list traversal used in figure 3, lines 7-14), or lack concurrency (AVL trees). To achieve the parallelism goal, we must express Anchors Hierarchy using only OpenMP or XMT-C idioms the compiler can transform for parallel execution at all loop levels. On the XMT this is a rich set of composable idioms including: general, Manhattan, and triangular loop collapse; reductions performed with atomic operations, hoisted out of loops, or performed in phases; and recurrences solved by parallel prefix. The compiler automatically performs scalar expansion, privatization, and allocates intermediate storage in order to break dependencies to distribute loops for parallelization.

Using these idioms, we are able to express all the loops necessary to implement Anchors Hierarchy through repeated sorting. Sorting can be implemented with no serial loops, meaning it can achieve the parallel coverage requirement, does not require sophisticated data structures, and eliminates problems related to load imbalance. This simplicity comes at a cost: the sorting method sorts all points every phase, however anchor-centric data structures minimize references to points which are not candidates for moving, reducing total work. We intend to better characterize the difference in total work with future experiments.

A. Sorting

This section briefly reviews the sorting techniques used in our implementation: counting sort and radix sort. These methods are *stable* sort methods, meaning keys of the same value are not reordered with respect to each other during sorting. This trait makes possible the conversion of the data dependency that serialized the inner loop (and is responsible for load imbalance) into a dependence resolved by sorting, which can be efficiently parallelized.

1) Counting Sort: Counting Sort is an efficient technique when the range of integer keys is known in advance and is small with respect to the number of points. Counting Sort has a complexity of $O(n + k)$, where n is the number of elements to sort and k is the number of unique keys [10]. The algorithm, illustrated in figure 7, proceeds by first histogramming the number of occurrences of each key, then a prefix operation stores the partial sums of counts as offsets for the sorted positions, and finally memory fills the length of the histogrammed sizes are performed at the offsets to generate the sorted list.

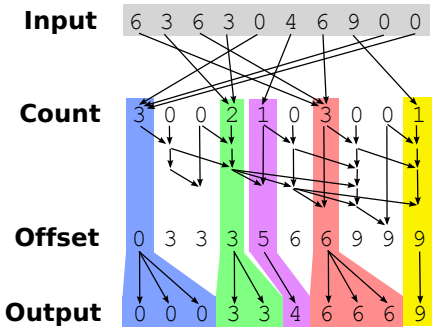


Fig. 7: Counting sort.

Each of the three phases can be performed in parallel: histograms are reductions which can be implemented using atomic increment instructions or scalable cyclic reductions, the recurrence to calculate offsets can be performed via parallel prefix, and the memory fill is a parallelizable Manhattan loop.

It is critical all the loops in counting sort are executed in parallel, particularly the nested loops, to ensure load balancing. The case in which all the elements have the same key must perform as well as the case of monotonically increasing keys. Counting sort, exposes triply-nested parallelism, however OpenMP limits us to 1 (occasionally 2) loop levels of parallelism, and the XMT compiler exploits only two loop nests of parallelism at a time. The XMT compiler can be coaxed into parallelizing either the inner two or outer two loops, allows us to have both implementations in the same program and dynamically select at runtime which version to call based on the expected loop trip counts. Our future work on this application will include characterizing which loop nests should be executed in parallel under which circumstances, and extend this manually collapsed OpenMP loops.

2) Radix Sort: Radix Sort, seen in figure 8, works by successively sorting by precision, beginning with the units of least precision and progressing towards the most significant units of precision [10]. Because radix sort iterates over positions, it can be used to sort values of any length or type. In our implementation, the stable sort used per pass is counting sort (section III-A1), which allows us to select the exact number of bits being sorted. Between Radix sort's iterative approach and counting sort's tunable range, it is possible to make explicit decisions about space-parallelism tradeoffs, research we intend to explore in the future.

Original	Pass 1	Pass 2	Pass 3
421976	438900	860034	008976
644203	656900	470940	127069
008976	644203	961413	421976
961413	961413	421976	438900
470940	476216	644203	470940
860034	860034	476216	476216
476216	470940	656900	644203
127069	127069	127069	656900
438900	421976	438900	860034
656900	008976	008976	961413

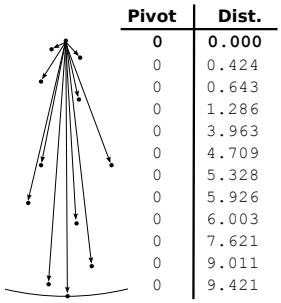
Fig. 8: Radix sort, radix = 2.

B. Anchors Formation Via Sorting

Instead of directly implementing a data structure describing a set of anchors, each with one pivot and zero or more points, we store three arrays: the pivot each point is associated with, the distance from the point to it's pivot, and a sorted rank. The data is first sorted by the distance field, then again by pivot. Because the sorting is stable, pivots in the resulting sorted lists can be indexed using parallel prefix techniques, and binary searches can be used to find points within anchors.

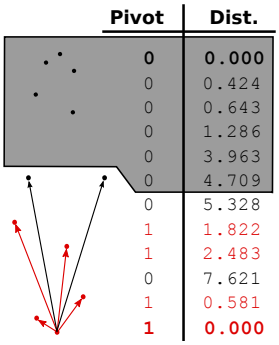
1) First and Second Anchor: The creation of the initial anchor occurs as it does in the serial algorithm (figure 9(a)). The second pivot is chosen as the point furthest from the initial pivot, and the half-plane between the two pivots is calculated. A binary search is made to find the range of points which are considered for associating the new pivot, and all the points are considered in parallel (figure 9(b)).

To restore the arrays to a sorted order which can be navigated by anchor and distance the entire



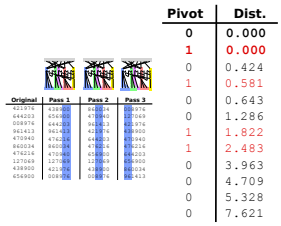
Pivot	Dist.
0	0.000
0	0.424
0	0.643
0	1.286
0	3.963
0	4.709
0	5.328
0	5.926
0	6.003
0	7.621
0	9.011
0	9.421

(a) The initial pivot is chosen at random and all the points are sorted by distance to the pivot.



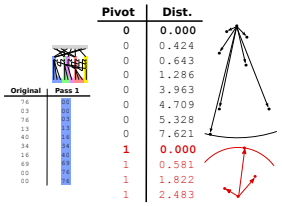
Pivot	Dist.
0	0.000
0	0.424
0	0.643
0	1.286
0	3.963
0	4.709
0	5.328
1	1.822
1	2.483
0	7.621
1	0.581
1	0.000

(b) The furthest point is selected as the new pivot, the half-plane between the two pivots determines the range of points to evaluate, points are reassigned to the nearer pivot.



Pivot	Dist.
0	0.000
1	0.000
0	0.424
1	0.581
0	0.643
0	1.286
1	1.822
1	2.483
0	3.963
0	4.709
0	5.328
0	7.621

(c) The pool of all points is sorted by distance (64 bits are sorted).



Pivot	Dist.
0	0.000
0	0.424
0	0.643
0	1.286
0	3.963
0	4.709
0	5.328
1	0.000
1	0.581
1	1.822
1	2.483

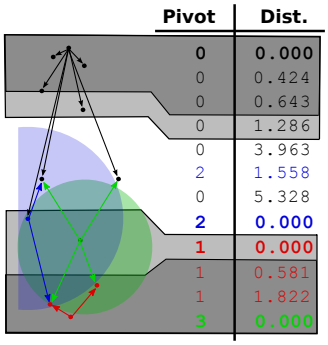
(d) The pool of all points is sorted by anchor ($\log k$ bits are sorted), permitting the anchors to be indexed via parallel prefix.

Fig. 9: Anchor formation via sorting. The first two anchors are created sequentially.

array is sorted by distance (figure 9(c)), and again by pivot (figure 9(d)). Sorting by distance involves all 64 bits of the double precision floating point numbers used for distances, however the number of pivots does not require 64 bits of precision. The radix sort allows sorting of only the $\log k$ bits needed to identify all the pivots.

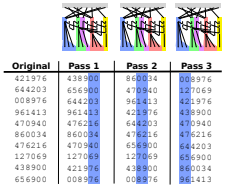
After sorting, the arrays can be indexed to find the locations of all the pivots. This is a parallel prefix loop the XMT compiler is able to parallelize but would be extremely challenging to perform in parallel using OpenMP. Future investigations will consider how anchor range indexing and other recurrences can be parallelized for OpenMP.

2) Parallel Anchor Addition: Our parallel algorithm adds multiple anchors simultaneously, trading work reduction via machine learning for additional



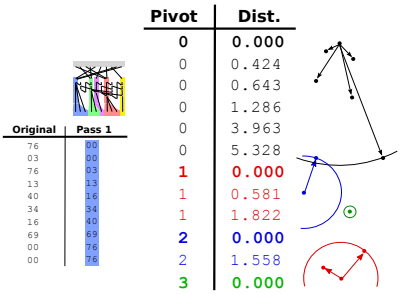
Pivot	Dist.
0	0.000
0	0.424
0	0.643
0	1.286
0	3.963
2	1.558
0	5.328
2	0.000
1	0.000
1	0.581
1	1.822
3	0.000

(a) The furthest point from each of the pivots are selected as new pivots, half-plane search spaces are found, and nearest new pivots are chosen.



Pivot	Dist.
0	0.000
2	0.000
1	0.000
3	0.000
0	0.424
1	0.581
0	0.643
0	1.286
2	1.558
1	1.822
0	3.963
0	5.328

(b) The pool of all points are sorted by distance. 64 bits are sorted.



Pivot	Dist.
0	0.000
0	0.424
0	0.643
0	1.286
0	3.963
0	5.328
1	0.000
1	0.581
1	1.822
2	0.000
2	1.558
3	0.000

(c) The points are sorted by anchor, and the pivots are indexed. Anchor formation is complete. $logk$ bits are sorted.

Fig. 10: Parallel anchor formation by addition of multiple pivots. These steps are repeated until the desired number of anchors have been formed.

concurrency. Because adding multiple anchors simultaneously has different semantics than adding one at a time, any new pivot selection method may be used. Understanding how new pivot selection affects the results will be part of our research.

In our current implementation, new pivots are selected from the furthest point in every extant

TABLE I: Properties of the corpora used in our experiments

	enwiki	1st 500K	filtered
Corpus Size	9.8GB	2.3GB	
Total words	1.74B	0.4B	0.335B
Unique words	13.3M	3.37M	
Number of documents	4571262	500000†	366613
Avg words/document	381	808	372
Avg unique words/doc	163	335	

†we extracted the first 500000 documents from enwiki

anchor. This is illustrated in figure 10(a): two new pivots are added and points are assigned to the pivot nearest them. The list is sorted first by distance (figure 10(b)), then by pivot (figure 10(c)), and the pivots are indexed. Additional anchors can be formed by repeating the steps in figures 10(a)-10(c). Note that pivot 3 has no points associated with it and at most only 3 new pivots can be added in the next phase.

IV. Preliminary Results

We ran all of the experiments reported here on our Cray XMT that has 128 processors, 1 TB of memory, and the Cray XMT 1.4.01 software.

A. English Wikipedia

We downloaded the wikipedia *pages-articles* file dated 30-Jan-2010[1]. A python library was used to extract the `<page>` content from non-redirect pages. The extracted content contained `html` tags and some wikipedia-specific encoding structures. We extracted as much as we could from this complex encoding. However, we are aware that this corpus has some non-words and this is borne out in the data showing so many unique words (See Table I). We selected only the first 500,000 documents from the over 4.5 million documents extracted from wikipedia. The high variance of document signature (non-zero vector entry) sizes caused load imbalance problems, so we arbitrarily filtered out all documents with fewer than 100 unique words and those documents with more than 10,000 unique words. Properties of the original and down-selected datasets are shown in Table I.

[1]The file we downloaded is *enwiki-20100130-pages-articles.xml.bz2* and was retrieved from *http://download.wikimedia.org/enwiki/20100130/*

B. Running times

We ran our code on the down-selected dataset and captured the time for computing the anchors; we do not include times for computing BoW. Figure 11 shows running times for varying number of processors on our Cray XMT.

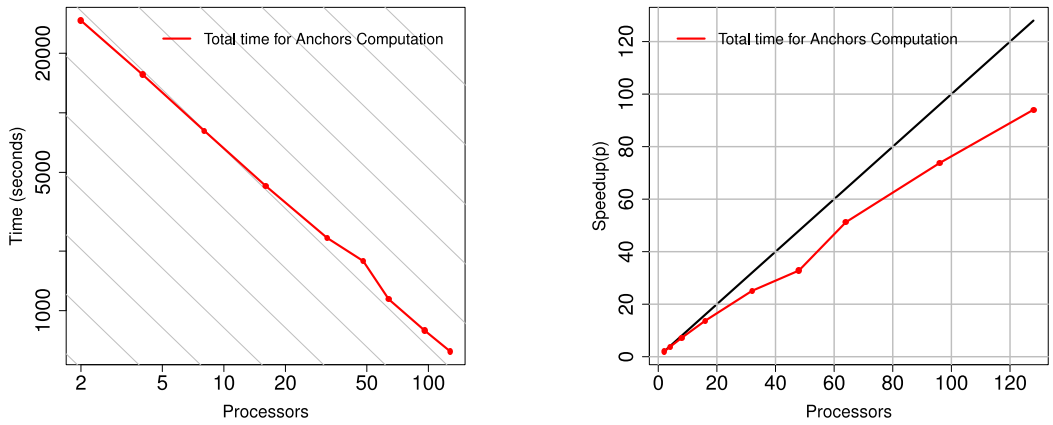
V. Future Work

The importance of triply nested parallel loops is made clear in Anchors Hierarchy where the idiom occurs twice in very different contexts. Load balanced execution is required in both instances because the work load inverts from the inner loop to outer. Furthermore, the nested parallelism is the source of the strong scaling which allows a larger system to solve a problem in less time than a smaller system, and larger problems to be run in the same amount of time as smaller problems by using larger systems.

The additional parallelism comes at the cost of redundant calculations — the anchors hierarchy is a machine learning technique, introducing multiple new anchors simultaneously reduces opportunities to eliminate distance calculations via the triangle inequality rule, increasing total work. It will be necessary to characterize how the redundant work is (not) overcome by parallelism and affects total execution time.

We have partially implemented a direct implementation of Moore's algorithm using linked listed data structures, we would like to complete this implementation and use it as a performance reference study to gauge the relative efficiency of our sorting version. The comparatively high single-threaded execution rate and small number of processors on a x86 system suggest the complex data structure technique may execute more effectively on x86 than the sorting technique.

The initial anchor is selected at random, a decision which might effect clustering quality. The sensitivity of the clusters formed with respect to initial pivot selection will be characterized. A related issue is our parallel implementation adds multiple anchors simultaneously, creating differences between serial and parallel execution. We will investigate how different pivot selection strategies affect execution performance and cluster quality.



(a) Running times (b) Speedup

Fig. 11: Performance/scalability on the Cray XMT architecture using the Down-selected English Wikipedia corpus

Although the XMT compiler performs extensive program transformations and rewrites, it fails to automatically parallelize some triply nested loop structures requiring transformations. We are interested in coaxing the compiler into performing these optimizations automatically, however, we will most likely need to perform them manually in any case in order to expose comparable nested parallelism for the OpenMP implementation.

ACKNOWLEDGMENT

This work was funded under the Center for Adaptive Supercomputing Software – Multithreaded Architectures (CASS-MT) at the Dept. of Energy's Pacific Northwest National Laboratory. Pacific Northwest National Laboratory is operated by Battelle Memorial Institute under Contract DE-ACO6-76RL01830.

REFERENCES

[1] N. Oikonomakou and M. Vazirgiannis, "A Review of Web Document Clustering Approaches," *Data Mining and Knowledge Discovery Handbook*, pp. 931–948, 2010.

[2] G. Salton, A. Wong, and C. S. Yang, "A vector space model for automatic indexing," *Commun. ACM*, vol. 18, no. 11, pp. 613–620, 1975.

[3] D. M. Blei, A. Y. Ng, M. I. Jordan, and J. Lafferty, "Latent dirichlet allocation," *Journal of Machine Learning Research*, vol. 3, p. 2003, 2003.

[4] D. J. Aldous, "Exchangeability and related topics," in *École d'été de probabilités de Saint-Flour, XIII—1983*, ser. Lecture Notes in Math. Berlin: Springer, 1985, vol. 1117, pp. 1–198. [Online]. Available: http://www.springerlink.com/content/c31v17440871210x/fulltext.pdf

[5] A. W. Moore, "The anchors hierarchy: Using the triangle inequality to survive high dimensional data," in *In Twelfth Conference on Uncertainty in Artificial Intelligence*. AAAI Press, 2000, pp. 397–405.

[6] G. Salton, E. A. Fox, and H. Wu, "Extended boolean information retrieval," *Commun. ACM*, vol. 26, pp. 1022–1036, November 1983. [Online]. Available: http://doi.acm.org/10.1145/182.358466

[7] E. L. Goodman, D. J. Haglin, C. Scherrer, D. Chavarria-Miranda, J. Mogill, and J. Feo, "Hashing strategies for the Cray XMT," in *Proceedings of the 24nd IEEE International Parallel and Distributed Processing Symposium*, 2010.

[8] Cray Inc., *Cray XMT Programming Environment User's Guide*. Cray Inc., 2010.

[9] J.-R. Sack and T. Strothotte, "A characterization of heaps and its applications," *Inf. Comput.*, vol. 86, pp. 69–86, May 1990. [Online]. Available: http://dx.doi.org/10.1016/0890-5401(90)90026-E

[10] D. E. Knuth, *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998.