Parallel Linkage

Hung-sik Kim
The Pennsylvania State University
University Park, PA 16802, USA
hungsik@psu.edu

Dongwon Lee
The Pennsylvania State University
University Park, PA 16802, USA
dongwon@psu.edu

ABSTRACT

We study the parallelization of the *(record) linkage* problem – i.e., to identify matching records between two collections of records, A and B. One of main idiosyncrasies of the linkage problem, compared to Database join, is the fact that once two records a in A and b in B are matched and merged to c, c needs to be compared to the rest of records in A and B again since it may incur new matching. This re-feeding stage of the linkage problem requires its solution to be iterative, and complicates the problem significantly. Toward this problem, we first discuss three plausible scenarios of inputs – when both collections are clean, only one is clean, and both are dirty. Then, we show that the intricate interplay between match and merge can exploit the characteristics of each scenario to achieve good parallelization. Our parallel algorithms achieve 6.55–7.49 times faster in *speedup* compared to sequential ones with 8 processors, and 11.15–18.56% improvement in *efficiency* compared to P-Swoosh.

Categories and Subject Descriptors: H.2.4 [Database Management]: Systems – concurrency, distributed databases, parallel databases

General Terms: Algorithm, Performance.

1. INTRODUCTION

Poor quality data is prevalent in databases due to a variety of reasons, including transcription errors, data entry mistakes, lack of standards for recording database fields, etc. To fix such errors, considerable recent work has focused on the **(record) linkage problem**, i.e., identify all matching records between two collections of records. Such a problem is also known as the *de-duplication* or *entity resolution* problem. The linkage problem frequently occurs in data applications (e.g., digital libraries, search engines, customer relationship management) and gets exacerbated when data are integrated from heterogeneous sources. For instance, a customer address table in a data warehouse may contain multiple address records that are all from the same residence, and thus need to be consolidated. For another example, imagine integrating two digital libraries, DBLP and CiteSeer. Since citations in two systems tend to have different formats, identifying all matching pairs is not straightforward.

Although the linkage problem has been studied extensively in various disciplines, by and large, the contemporary approaches have focused on how to identify "`matching`" records "`faster`" using a "`better`" distance function. For instance, a nested-loop style algorithm, $\mathcal{A}$, with $O(mn)$ running time can solve the linkage problem as follows (assuming a distance function, $dist()$, and a preset threshold, θ):

for each record $a_i (\in A)$
  for each record $b_j (\in B)$
    if $dist(a_i, b_j) < \theta$, then $a_i \approx b_j$

Due to its quadratic nature, however, when both inputs A and B have a large number of records, the naive approach becomes prohibitively expensive. To remedy this problem, people have proposed many improvements – e.g., faster linkage approaches such as blocking [21] or computationally efficient distance functions such as upper-bound matching [15]. Toward this scalability problem, however, we take a different approach, and study how to make a linkage algorithm "parallel." Therefore, in this paper, we do *not* consider issues like which distance function, $dist()$, to use or how to add blocking/indexing to $\mathcal{A}$ (as in blocked nested-loop or indexed join) and leave that as future work.

Another important aspect of the linkage problem – how to *merge* records that are found to be matching – has been largely ignored until recently when SERF project studied it explicitly [14]. This novel perspective emphasized that two main differences between the linkage problem and Database join problem are: (1) the linkage problem has both matching and merging steps tangled while the join problem has only the matching step, and (2) the linkage problem often involves the entire set of (long string) attributes in a record in the matching step while the join problem often focuses on a few (numeric or short string) attributes.

To take these points into consideration, the linkage problem that we consider in this paper can be defined as follows:

> **Linkage Problem**: Given two collections of compatible records, $A=\{a_1, ..., a_m\}$ and $B=\{b_1, ..., b_n\}$, do: (1) identify and merge all matching (i.e., $\approx$) record pairs (a_i,a_j), (b_i,b_j), or (a_i,b_j), and (2) create a merged collection $C=\{c_1, ..., c_k\}$ of A and B such that $\forall c_i, c_j \in C$, $c_i \not\approx c_j$.

Note that neither A nor B itself is assumed to be *clean* (to be

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CIKM'07, November 6–8, 2007, Lisboa, Portugal.
Copyright 2007 ACM 978-1-59593-803-9/07/0011 ...$5.00.

defined in Definition 2) – i.e., there may be two matching records in it. Therefore, we investigate three scenarios – when both collections are clean, when only one is clean, and when both are dirty. Furthermore, we show that the intricate interplay between matching and merging steps can exploit the characteristics of each scenario to achieve good parallelization. The intuition of our algorithms is that if: (1) a_i is deemed to be a duplicate of b_j, and (2) an input collection B is a set, not a bag (i.e., clean), then one does not need to check if a_i is a duplicate of b_{j+1}, ..., b_n in the algorithm $\mathcal{A}$. Depending on the relationship between a_i and b_j (i.e., a_i contains b_j, b_j contains a_i, a_i is identical to b_j, or a_i is overlapping with b_j), this intuition can be exploited differently.

Our contributions are as follows: (1) We formally introduce the linkage problem with separate match and merge steps, and exploit them to have better sequential linkage framework for three scenarios; (2) We extend sequential linkage algorithms to parallel ones under three scenarios such that redundant computation and overhead among multiple processors are minimized; (3) Our proposals are evaluated using citation data sets with a variety of characteristics. Our parallel algorithms achieve 6.55–7.49 times faster in *speedup* compared to sequential ones with 8 processors, and 11.15–18.56% improvement in *efficiency* compared to P-Swoosh.

2. BACKGROUND

Consider two records, r and s, with $|r|$ columns each, where $r[i]$ $(1 \leq i \leq |r|)$ refers to the i-th column of the record r. Further, let us assume that corresponding columns of r and s have compatible domain types: $dom(r[i]) \sim dom(s[i])$.

Definition 1 (Record Matching) *When two records, r and s, are deemed to refer to the same real-world entity, both are said* matching*, and written as $r \approx s$ (otherwise $r \not\approx s$).* □

Note that how one determines if two records refer to the same real-world entity or not is *not* the concern of this paper. Assuming the existence of such oracle, we focus on the parallelization of the linkage problem instead. In practice, however, the matching of two records can be often determined by distance or similarity functions. For instance, one may use the cosine angle of token sets of r and s (i.e., cosine similarity) to determine the match of r and s. Or, one may use the ratio of intersected vs. unioned q-gram tokens of two records (i.e., jaccard similarity) for the same purpose.

When two records, r and s, are matching (i.e., $r \approx s$), four relationships, as illustrated in Figure 1, can occur: (1) $r \sqsupseteq s$: all information of s appears in r, (2) $r \sqsubseteq s$: all information of r appears in s, (3) $r \equiv s$: information of r and s is identical (i.e., $r \sqsupseteq s \wedge r \sqsubseteq s$), and (4) $r \oplus s$: neither (1) nor (2), but the overlap of information of r and s is beyond a threshold θ. Note that to be a flexible framework we do *not* tie the definitions of the four relationships to a particular notion of *containment* or *overlap*. Instead, we assume that the containment or overlap of two records can be further specified by users or applications. Let us assume the existence of two such functions: (1) **contain(r,s)** returns True if r contains s, and False otherwise, and (2) **match(r,s)** returns True (i.e., one of the four inter-record relationships) or False for non-matching. We assume that match(r,s) is implemented using contain(r,s) function internally (e.g., if both contain(r,s) and contain(s,r) return True, then match(r,s) returns $r \equiv s$).

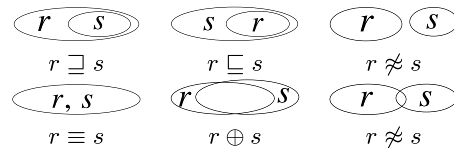


Figure 1: Inter-record relationships.

Symbol	Meaning
A, B	two input collections
m and n	size of A and B, i.e., $m = \|A\|$, $n = \|B\|$
r or a_i	a record in A
s or b_j	a record in B
c_{ij} or $c_{i,j}$	a merged record from a_i and b_j
θ	threshold for $r \oplus s$
contain(a_i,b_j)	returns True if a_i contains b_j, or False
match(a_i,b_j)	returns $\sqsupseteq$, $\sqsubseteq$, $\equiv$, $\oplus$, or $\not\approx$
merge(a_i,b_j)	returns c_{ij}

Table 1: Summary of notations.

Example 1. For a table with five columns, consider the following records: r_1:("a",−,−,−,−), r_2:(−,"b", −, −, −), r_3:("a","b","c", −, −), r_4:(−,"b","c","d",−), and r_5:(−, −, −,"d","e"). Further, let us assume two function: (1) the containment of two records is determined by the containment of token sets of two records, and (2) the overlap of two records is measured by the average jaccard similarity of two corresponding columns of two records with $\theta = 0.3$. Then, $r_1 \sqsubseteq r_3$ holds since {a} ⊆ {a, b, c}, $r_2 \sqsubseteq r_3$ holds since {b} ⊆ {a, b, c}, and $r_2 \sqsubseteq r_4$ since {b} ⊆ {b, c, d}. In addition, jaccard(r_3,r_4)= $\frac{0+1+1+0+0}{5} = 0.4 > \theta$ and jaccard(r_4,r_5)=$\frac{0+0+0+1+0}{5} = 0.2 < \theta$. Therefore, both $r_3 \oplus r_4$ and $r_4 \not\approx r_5$ hold. □

When two records r and s are matching (i.e., $r \sqsubseteq s$, $r \sqsupseteq s$, $r \equiv s$, or $r \oplus s$), one can merge them to get a record with more (or better) information. Again, how exactly the merge is implemented is not the concern of this paper. We simply refer to a function that merges r and s to get a new record w as **merge(r,s)**.

Example 2. For instance, like [14], if one uses the set union operator, ∪, as the merge function for Example 1, then merge(r_3,r_4) would generate a new record r_{34}: ("a","b","c", "d", −), while merge(r_1, r_3) would generate r_{13}=r_3:("a", "b", "c", −, −) since $r_1 \sqsubseteq r_3$. □

Definition 2 (Clean vs. Dirty) *When a collection A has no matching records in it, it is called* clean*, and* dirty *otherwise. That is, (1) A is clean iff $\forall r, s \in A$, $r \not\approx s$, and (2) A is dirty iff $\exists r, s \in A$, $r \approx s$.* □

Table 1 summarizes the notations.

Related Work. The general linkage problem has been known as various names – record linkage (e.g., [8, 3]), identity uncertainty (e.g., [17]), merge-purge (e.g., [11]), citation matching (e.g., [16]), object matching (e.g., [4]), entity resolution (e.g., [18]), and approximate string join (e.g., [10]) etc. Since the focus of our parallel linkage is orthogonal to many of these works, in this section, we survey a few recent representative works only.

$A \setminus B$	Clean	Dirty
Clean	Sections 3.1 and 4.1	Sections 3.2 and 4.2
Dirty	Sections 3.2 and 4.2	Sections 3.3 and 4.3

Table 2: Taxonomy.

Unlike the traditional methods exploiting textual similarity, Constraint-Based Entity Matching (CME) [20] examines "semantic constraints" in an unsupervised way. They use two popular data mining techniques, Expectation-Maximization (EM) and relaxation labeling for exploiting the constraints. [1] presents a generic framework, *Swoosh* algorithms, for the entity resolution problem. The recent work by [7] proposes an iterative de-duplication solution for complex personal information management. Their work reports good performance for its unique framework where different deduplication results mutually reinforce each other (e.g., the resolved co-author names are used in resolving venue names).

Another recent trend in linkage problem is to exploit additional information beyond simple string comparison. For instance, [12] presents a relationship-based data cleaning (RelDC) which exploits context information for entity resolution, or [6] proposes a generic semantic distance metric between two terms using the page counts from the Web.

Parallel database join has been well studied (e.g., [19]). However, as mentioned in Section 1, parallel linkage has distinct characteristics, making the application of parallel join solutions non-trivial. In recent years, parallel linkage has been studied in P-Febrl [5], D-Swoosh [2], and P-Swoosh [13]. P-Febrl is the parallelization model by Python module *Pypar* but no detailed algorithms are shown. Both D-Swoosh and P-Swoosh, parallel versions of Swoosh [1], are implemented by Java emulator, and runs in dual core processors. In parallel structures, D-Swoosh uses the *task graph model* while P-Swoosh uses the *master-slave model*. Our algorithm, implemented in distributed MATLAB, runs in real parallel environment (while P-Swoosh runs only in simulated environment). Our parallel solutions use the task graph model to keep simple control of load balancing. All of works can adapt any ER algorithm as a matching function.

3. SEQUENTIAL LINKAGE

We investigate three scenarios depending on types of input sets, as shown in Table 2: (1) *clean vs. clean*: This scenario is relevant when two already-clean data sources are integrated, (2) *dirty vs. clean*: Consider a search engine that has a clean data set A, but its crawler fetches new dirty data set B every day. In this case, not only B may have matching records in it, there can be new matching pairs between A and B, (3) *dirty vs. dirty*: When one has two dirty sets, one can clean each dirty set independently and apply clean vs. clean scenario. However, as we will present, one may be able to improve the linkage by matching two dirty sets directly. The scenario of cleaning single dirty set A (i.e., self cleaning) will be shown to be covered by dirty-clean or dirty-dirty cases easily.

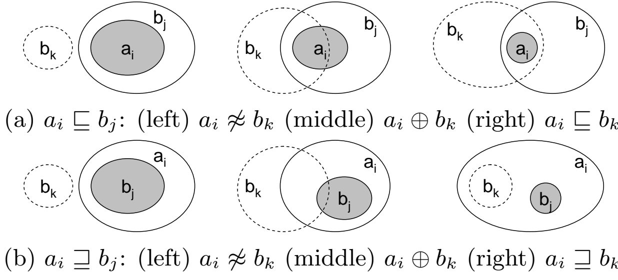
3.1 Clean vs. Clean

Recall that unlike database join, in the linkage problem, if two records a_i and b_j match, then a merged record c_{ij} (= merge(a_i,b_j)) is created and re-feeded into A and B. However, depending on the type of matching, one can do further saving. Suppose we use the naive algorithm $\mathcal{A}$ in Section 1 for its simplicity. Consider four records: a_i, a_l in A ($i < l \leq m$), b_j, b_k in B ($j < k \leq n$), and two sets E (to hold an instance of two identical records) and C (to hold merged record c_{ij} of a_i and b_j). Then, if:

- $a_i \not\approx b_j$: Proceed to the next match(a_i,b_{j+1}).
- $a_i \equiv b_j$: First, add a_i to E. Then, remove a_i from A and b_j from B. Since B is a clean set, by definition, there cannot be any matching records to b_j in B. Since a_i is identical to b_j, in addition, there cannot be any matching records to a_i in B, either. Therefore, we do not need to compute: match(a_i,b_k). Symmetrically, since A is also a clean set and a_i is identical to b_j, there cannot be any matching records to b_j in A, either, and thus we do not need to compute match(a_l,b_j). At the end, therefore, $m - i + n - j$ times of computation of match() function is saved. Finally, proceed to the next match(a_{i+1},b_1).
- $a_i \sqsubseteq b_j$: First, remove a_i from A. Between a_i and b_k, two relationships, $b_k \sqsubseteq a_i$ and $a_i \equiv b_k$, cannot occur since B is a clean set (e.g., if $b_k \sqsubseteq a_i \wedge a_i \sqsubseteq b_j$, then $b_k \sqsubseteq b_j$ by transitivity, but since B is a clean set, $b_k \not\sqsubseteq b_j$, leading to a contradiction). However, the other three relationships can occur, as illustrated in Figure 2(a). Note that if the only possible relationship between a_i and b_k was $a_i \not\approx b_k$, then we could have skipped the computation of match(a_i,b_k). However, since there are three possibilities, we may not entirely skip match(a_i,b_k). However, note that in Figure 2(a), (1) if $a_i \sqsubseteq b_k$, then even if we do compute match(a_i,b_k), it does not generate any new record since merge(a_i,b_k)=b_k. Therefore, this case can be ignored, and (2) $a_i \oplus b_k$ case is rare in practice. Since $b_j \not\approx b_k$, if $a_i \sqsubseteq b_j$, then most likely $a_i \not\approx b_k$ holds, although an extreme case like Figure 2(a) (middle) can happen. Therefore, we can skip the entire computation of match(a_i,b_k) with the risk of rare false negatives. In the experimentation, we empirically show that the risk is quite low[1]. Finally, proceed to the next match(a_{i+1},b_1).
- $a_i \sqsupseteq b_j$: First, remove b_j from B. between a_i and b_k, two relationships, $a_i \sqsubseteq b_k$ and $a_i \equiv b_k$, cannot occur since B is a clean set, but the other three relationships can occur, as illustrated in Figure 2(b). Since there are three possibilities between a_i and b_k, we may not skip the computation of match(a_i,b_k). Finally, proceed to the next match(a_i,b_{j+1}).
- $a_i \oplus b_j$: First, remove both a_i from A and b_j from B. Then, add c_{ij} (= merge(a_i,b_j)) to C. Finally, proceed to the next match(a_i,b_{j+1}).

The iterative sequential linkage algorithm for clean-clean case, referred to as **s-CC**, is shown in Algorithm 2 that terminates when no more merge() occurs (line 1). The main functionality of **s-CC** using the five cases of inter-record relationships is captured in **s-CC-single** of Algorithm 1. At line 1 of **s-CC-single**, both $m = |A|$ and $n = |B|$ continue to shrink as records in A or B are removed. The function

[1]By setting the threshold for overlap, θ, as substantially high or low, we can decrease the risk of false negatives even further.



(a) $a_i \sqsubseteq b_j$: (left) $a_i \not\approx b_k$ (middle) $a_i \oplus b_k$ (right) $a_i \sqsubseteq b_k$

(b) $a_i \sqsupseteq b_j$: (left) $a_i \not\approx b_k$ (middle) $a_i \oplus b_k$ (right) $a_i \sqsupseteq b_k$

Figure 2: Six possible relationships for a_i, b_j, and b_k when $b_j \not\approx b_k$ (i.e., B is clean).

Input : Two non-empty *clean* lists A and B
Output : Intermediate lists A', B' and a *clean* list C
/* E is a temporary list to contain equality records */
$i \leftarrow j \leftarrow 1$, $A' \leftarrow B' \leftarrow C \leftarrow E \leftarrow \emptyset$;
1 **while** $i \leq |A|$ and $j \leq |B|$ **do**
  switch match(a_i,b_j) **do**
    case $a_i \equiv b_j$
      add a_i to E; remove a_i from A and b_j from B;
      $i \leftarrow i+1$, $j \leftarrow 1$;
    case $a_i \sqsubseteq b_j$
      remove a_i from A; $i \leftarrow i+1$, $j \leftarrow 1$;
    case $a_i \sqsupseteq b_j$ remove b_j from B; $j \leftarrow j+1$;
    case $a_i \oplus b_j$
      remove a_i from A and b_j from B;
2     $C \leftarrow$ s-merge(c_{ij},C); $i \leftarrow i+1$, $j \leftarrow 1$;
    case $a_i \not\approx b_j$ $j \leftarrow j+1$;
  if $j > |B|$ **then** $i \leftarrow i+1$, $j \leftarrow 1$;
$A' \leftarrow A$, $B' \leftarrow B \cup E$; return (A', B', C);

Algorithm 1: s-CC-single.

s-merge(c_{ij},C), whose details are omitted, merges a record c_{ij} into a clean list C, ensuring that resulting list C be still clean by comparing c_{ij} to all records in C.

3.2 Dirty vs. Clean

The detailed procedure for dirty-clean case, **s-DC**, is shown in Algorithm 4 that uses Algorithm 3 as a sub-step. When only one collection, A, is *dirty*, one can use the other clean collection, B, as the final merged clean set C to minimize space cost. Therefore, when either $\sqsubseteq$ or $\sqsupseteq$ relationship occurs, the record can be simply removed from one collection (lines 1 and 2 of **s-DC-single**). Similarly, when $\oplus$ relationship occurs (line 3), both original records, a_i and b_j, are removed and the new matched record c_{ij} is added to the dirty collection C. After the iteration, when a_i is not matched to any records b_j from B (line 3), it becomes safe to move a_i to the clean collection, B. From the next iteration, this newly-moved record a_i will be compared to the rest of records A. This step is necessary since A was dirty. When no more merge occurs in the line 3 of Algorithm 3

Input : Two non-empty *clean* lists A and B
Output : A single merged *clean* list C
$A' \leftarrow B' \leftarrow C \leftarrow E \leftarrow \emptyset$;
1 **while** $A \neq \emptyset$ **do**
  $(A', B', C) \leftarrow$ **s-CC-single**(A, B); $A \leftarrow C$; $B \leftarrow A' \cup B'$;
$C \leftarrow B$; return C;

Algorithm 2: s-CC.

Input : Non-empty *dirty* list A and *clean* list B
Output : An intermediate list B' and merged *dirty* list C
$i \leftarrow j \leftarrow 1$, $C \leftarrow \emptyset$;
while $i \leq |A|$ and $j \leq |B|$ **do**
  switch match(a_i,b_j) **do**
    case $a_i \equiv b_j$ or $a_i \sqsubseteq b_j$
1   remove a_i from A; $i \leftarrow i+1$, $j \leftarrow 1$;
2   **case** $a_i \sqsupseteq b_j$ remove b_j from B; $j \leftarrow j+1$;
    case $a_i \oplus b_j$
      remove a_i from A and b_j from B;
3   add c_{ij} to C; $i \leftarrow i+1$, $j \leftarrow 1$;
    case $a_i \not\approx b_j$ $j \leftarrow j+1$;
  if $j > |B|$ *or* $B = \emptyset$ **then**
    add a_i to B; remove a_i from A;
    $i \leftarrow i+1$, $j \leftarrow 1$;
$B' \leftarrow B$; return (B', C);

Algorithm 3: s-DC-single.

Input : Non-empty *dirty* list A and *clean* list B
Output : One merged *clean* list C
1 **while** $A \neq \emptyset$ **do**
  $(B', C) \leftarrow$ **s-DC-single**(A,B); $A \leftarrow C$; $B \leftarrow B'$;
$C \leftarrow B$; return C;

Algorithm 4: s-DC.

(i.e. A is empty in the line 1 of Algorithm 4) the algorithm terminates.

By using **s-DC**, note that one can clean a single dirty collection A. That is, by moving the first record from A to B, one can turn the problem into the sequential linkage of dirty-clean case. As a syntactic sugar, let us call this algorithm as **s-self**, shown in Algorithm 5. Then, another way to implement the sequential linkage for dirty-clean case to use **s-self** and **s-CC** – i.e., clean the dirty collection A using **s-self** first and apply the sequential linkage for clean-clean case. To distinguish from the **s-DC** of Algorithm 4, we denote this implementation as **s-DC**$_{self}$. Algebraically, the following holds: **s-DC**$_{self}(A,B) \equiv$ **s-CC**(**s-self**(A),B).

Note that algorithms **s-DC** and **s-DC**$_{self}$ behave differently depending on the level of "dirty-ness" within A or between A and B. For instance, consider three records, a_i, $a_k \in A$ and $b_j \in B$. Suppose the following relationship occurs: $a_i \oplus a_k \sqsubset b_j$. Then, using **s-DC**$_{self}$, $a_i \oplus a_k$ will be compared again with other records in A. However, using **s-DC**, a_i and a_k will be removed, saving $|A|$ number of comparisons. On the other hand, for instance, assume that $a_i \approx a_k$, $a_i \not\approx b_j$, and $a_k \not\approx b_j$. Using **s-DC**$_{self}$, there is only one comparison after $a_i \approx a_k$ is made. However, using **s-DC**, both a_i and a_k are compared to b_j before $a_i \approx a_k$ occurs, increasing the number of comparisons. In general, if the number of matches in A is significantly higher than that between A and B, then **s-DC**$_{self}$ is expected to perform better.

3.3 Dirty vs. Dirty

Input : A non-empty *dirty* list A
Output : A non-empty *clean* list C
$B \leftarrow C \leftarrow \emptyset$; move a_1 from A to B; $C \leftarrow$ **s-DC**(A,B); return C;

Algorithm 5: s-self.

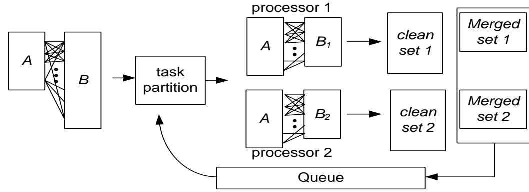


Figure 3: Parallel linkage model with 2 processors.

Since neither collection A or B is clean, more comparisons are needed for dirty-dirty case. By using sequential linkage algorithms for clean-clean or dirty-clean cases, we propose three variations, referred to as **s-DD1**, **s-DD2**, and **s-DD3**, as follows:

1. **s-DD1**$(A,B) \equiv$ **s-DC**$(A,$**s-self**$(B))$
2. **s-DD2**$(A,B) \equiv$ **s-CC**(**s-self**(A),**s-self**$(B))$
3. **s-DD3**$(A,B) \equiv$ **s-self**$(A \cup B)$

The different behaviors of variations will be evaluated experimentally.

4. PARALLEL LINKAGE

The high-level overview of parallel linkage using 2 processors is illustrated in Figure 3. Parallel linkage is a distributed algorithm to perform matching and merging concurrently. To achieve this, either data or task needs to be partitioned and distributed to multiple processors. In an ideal parallel model, tasks are evenly distributed among all processors. However, in reality, such an even partition is not trivial since a task cannot be measured easily before it actually executes. In our setting, therefore, we estimate the number of record comparisons by the size of data, and use the data partition model to *simulate* the task partition model.

Given two inputs, A and B with $|A| \leq |B|$, the gist of our parallel algorithms is that each processor P_i has a replicated A and a partition of B, called B_i. At each P_i, then, an appropriate sequential linkage is done separately (e.g., **s-CC-single**(A,B_i) for clean-clean case). Once intra-processor cleanness is ensured using sequential linkage, next, inter-processor cleanness needs to be addressed. Therefore, the outputs of sequential linkage at P_i are then properly shipped and compared to the rest of data at the other processors. This process repeats until no more merge() occurs at any processors.

To make the presentation simpler, we assume that each processor P_i has a local queue, Q_i, while there is a single global queue Q_G. With an efficient implementation using the linked list or priority queue, we assume that operations such as *enqueue*(a,Q), *enqueue*$(\{a_1, a_2\},Q)$ (= *enqueue*$(a_2,$*enqueue*$(a_1,Q)))$, and *dequeue*(Q) are efficiently supported for all queue data structures. Furthermore, we assume the following functions:

- The *partition*(Q_G) function, shown in Algorithm 6, takes a global queue Q_G (containing a "list" of list of records) as input, dequeues a list, say B, from Q_G, partitions it to k pieces of B_1, ..., B_k, and ships both the remainder of Q_G and B_i to each P_i.

```
Input  : A queue, Q_G, containing a list of list of records
Result : At each P_i, a sub-list B_i and a local queue Q_i are set
B ← dequeue(Q_G);
partition B to k sub-lists: B_1, ..., B_k;
ship B_i and Q_G to P_i;
foreach P_i (1 ≤ i ≤ k) do Q_i ← Q_G
```

Algorithm 6: partition.

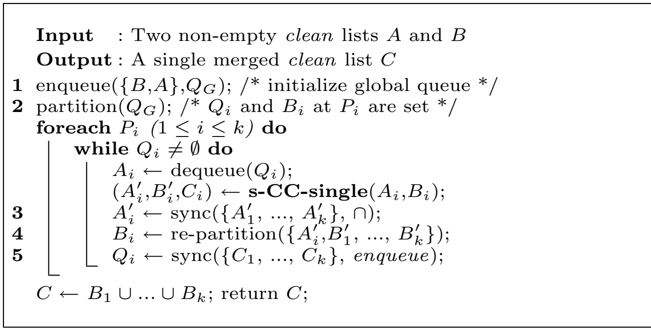
- The *sync*$(\{A_1, ..., A_k\}$, *op*$)$ function synchronizes values of input sets of all processors with respect to the specified operator, *op*. For instance, with A_1={1,3,4} and A_2={1,2,4,5}, *sync*$(\{A_1,A_2\}, \cap)$ would synchronize two sets by applying $\cap$ to yield a set {1,4}. However, *sync*$(\{A_1,A_2\},\cup)$ would yield a set {1,2,3,4,5}. To avoid communication cost among processors, in the implementation, one exchanges only indexes of input sets, instead of actual data sets.
- Recall that each processor P_i initially has a replicated A and partitioned B_i data sets. After a sequential linkage runs at each processor, depending on the occurrence of match() and merge(), results of each linkage may be different. In such a case, to ensure even distribution of data/task among all processors, one needs to re-partition B_i again. The *re-partition*$(\{A, B_1, ..., B_k\})$ re-partitions B_i while considering A so that data are evenly distributed among A and all B_is. For example with two processors, if A={1,3,4} and B_1={5,6}, B_2={7,8,9}, then *re-partition*$(\{A,B_1,B_2\})$ results in A={1,3,4}, B_1={5,6,1,3}, and B_2={7,8,9,4}. This will do load-balancing by adjusting number of partitioned records of B_i.

4.1 Clean vs. Clean

The parallel linkage for two clean inputs, referred to as **p-CC** in Algorithm 7, is the parallelization of the sequential linkage **s-CC**. Suppose there are k processors, $P_1, ..., P_k$. Once the data set B is partitioned to B_i and shipped to each processor (lines 1 and 2), at each processor P_i, a single iteration of sequential linkage for clean-clean, **s-CC-single**, is applied to generate a clean set C_i and two intermediate sets of A'_i and B'_i. Note that the initial input data set A was replicated to all processors. However, each intermediate set of A'_i may be different since A is compared to different piece of B. Therefore, to avoid redundant comparison, we needs to synchronize all intermediate A'_i from all processors (line 3). Similarly, intermediate B'_i at each processor may have different values after **s-CC-single**. To increase the efficiency of parallel linkage, therefore, one needs to re-distribute B'_i across all processors (line 4). Finally, all clean sets C_i generated from **s-CC-single** are gathered and re-feeded into the local queue Q_i (line 5). This step is necessary since a clean set of P_1 still needs to be compared to intermediate sets in P_1 as well as in another processor P_2.

The algorithm to clean n clean input sets, termed as **p-CC-multi**, can be straightforwardly made by extending the **p-CC** that links "two" clean input sets (i.e., at line 1 of **p-CC**, all n input clean sets need to be enqueued to Q_G).

Example 3. Using Figure 4, let us illustrate how merged sets can have inter- and intra-comparisons to intermediate sets iteratively in **p-CC**. Consider two clean input sets,



Input : Two non-empty *clean* lists A and B
Output : A single merged *clean* list C
1 enqueue($\{B,A\}$,Q_G); /* initialize global queue */
2 partition(Q_G); /* Q_i and B_i at P_i are set */
foreach P_i $(1 \le i \le k)$ **do**
  while $Q_i \ne \emptyset$ **do**
    $A_i \leftarrow$ dequeue(Q_i);
    $(A'_i, B'_i, C_i) \leftarrow$ **s-CC-single**(A_i,B_i);
3   $A'_i \leftarrow$ sync($\{A'_1, ..., A'_k\}, \cap$);
4   $B_i \leftarrow$ re-partition($\{A'_i, B'_1, ..., B'_k\}$);
5   $Q_i \leftarrow$ sync($\{C_1, ..., C_k\}$, *enqueue*);
$C \leftarrow B_1 \cup ... \cup B_k$; return C;

Algorithm 7: p-CC.

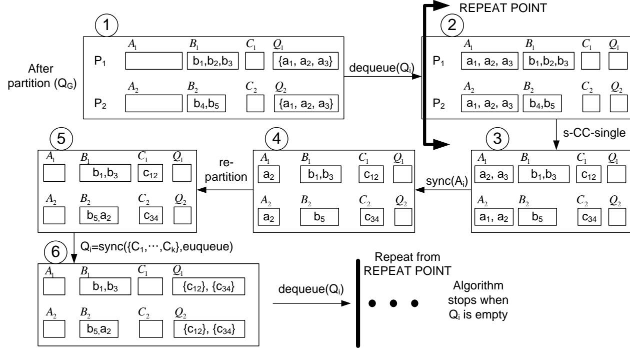


Figure 4: Single iteration of p-CC.

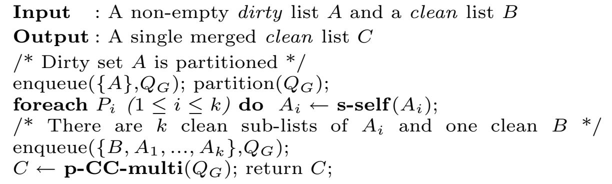
$A = \{a_1, a_2, a_3\}$ and $B = \{b_1, b_2, b_3, b_4, b_5\}$, and two processors, P_1 and P_2. Furthermore, let us assume that only two merge() occur: $c_{12} = a_1 \oplus b_2$ and $c_{34} = a_3 \oplus b_4$. Since $|B| > |A|$, B is partitioned to $B_1 = \{b_1, b_2, b_3\}$ in P_1 and $B_2 = \{b_4, b_5\}$ in P_2 and A is replicated to Q_1 and Q_2 (first box in Figure 4). Now, after dequeue(Q_i) is done (second box) and **s-CC-single** runs at each processor (third box), we have: $A'_1 = \{a_2, a_3\}$, $B'_1 = \{b_1, b_3\}$, and $C_1 = \{c_{12}\}$ at P_1, and $A'_2 = \{a_1, a_2\}$, $B'_2 = \{b_5\}$, and $C_2 = \{c_{34}\}$ at P_2 (third box). Then, sync($\{A'_1, A'_2\}, \cap$) of line 3 in **p-CC** yields $A'_1 = A'_2 = \{a_2\}$ (fourth box). Since B_1 and B_2 have different left-over and $|B_2| < |B_1|$, by re-partition on line 4 in **p-CC**, a_1 is added to B_2 to make $B_1 = \{b_1, b_3\}$ and $B_2 = \{b_5, a_2\}$ (fifth box). On line 5 in **p-CC**, if C_i is not empty, algorithm enqueues C_i to all local queues. Then, both Q_1 and Q_2 contain both C_1 and C_2 by synchronizing C_1 and C_2 among other processors (sixth box). This steps repeat until a queue Q_i is empty. □

4.2 Dirty vs. Clean

We propose two different parallel schemes, named as **p-DC**$_{self}$ and **p-DC**, similar to two sequential schemes of **s-DC**$_{self}$ and **s-DC**, respectively.

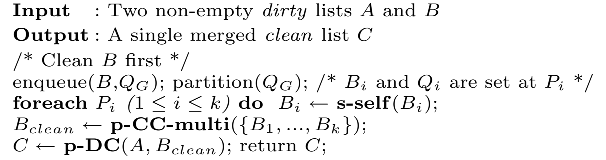
In **p-DC**$_{self}$ of Algorithm 8, first *dirty* input set A is partitioned to A_i and distributed to each processor. Then, each A_i is separately cleaned by applying **s-self** at each processor. At this point, k *clean* sub-lists and one *clean* input list, B, remain. Then, all these clean sub-lists can be gathered and cleaned, including B, by **p-CC-multi**.

In **p-DC** (algorithm not shown for space), like **s-DC**$_{self}$, the *dirty* list A is partitioned to k sub-lists, $A_1, ..., A_k$ at each processor. However, unlike **s-DC**$_{self}$, the *clean* list B is also shipped to each processor. Then, dirty-clean case



Input : A non-empty *dirty* list A and a *clean* list B
Output : A single merged *clean* list C
/* Dirty set A is partitioned */
enqueue($\{A\}$,Q_G); partition(Q_G);
foreach P_i $(1 \le i \le k)$ **do** $A_i \leftarrow$ **s-self**(A_i);
/* There are k clean sub-lists of A_i and one clean B */
enqueue($\{B, A_1, ..., A_k\}$,Q_G);
$C \leftarrow$ **p-CC-multi**(Q_G); return C;

Algorithm 8: p-DC$_{self}$.



Input : Two non-empty *dirty* lists A and B
Output : A single merged *clean* list C
/* Clean B first */
enqueue(B,Q_G); partition(Q_G); /* B_i and Q_i are set at P_i */
foreach P_i $(1 \le i \le k)$ **do** $B_i \leftarrow$ **s-self**(B_i);
$B_{clean} \leftarrow$ **p-CC-multi**($\{B_1, ..., B_k\}$);
$C \leftarrow$ **p-DC**(A, B_{clean}); return C;

Algorithm 9: p-DD1.

of sequential linkage algorithm, **s-DC-single**(A_i, B) is executed at each processor. Once data at each processor is cleaned, multiple clean lists from all processors can be gathered and cleaned by **p-CC-multi**.

4.3 Dirty vs. Dirty

We propose two parallel linkage schemes to handle two dirty lists: (1) **p-DD1**, parallelization of **s-DD1**, cleans one dirty set first, then apply **p-DC**, while **p-DD2**, parallelization of **s-DD2**, attempts to clean both sets at the same time and apply **p-CC**.

In **p-DD1** of Algorithm 9, first, data set B is partitioned to k pieces and cleaned by **s-self** at each processor. When k clean sub-lists of B are created, they are merged back via a queue and cleaned by the parallel linkage solution **p-CC-multi**. After B is cleaned and stored in B_{clean}, then, we apply **p-DC** to get a merged clean list of C. In the **p-DD2** scheme (algorithm not shown for space), each input set A and B are separately partitioned and cleaned by **s-self**, generating $2k$ clean sub-lists of A and B. At the end, all these sub-lists are cleaned by **p-CC-multi**.

5. ANALYSIS

In this section, we present a few important properties of the representative sequential and parallel linkage algorithms. First, we show the termination of **s-self** algorithm since it represents the worst case of input data.

Lemma 1. *The **s-self**(A) algorithm terminates.* ■

PROOF. The algorithm will stop when $|A| = 0$. Note that the output of match() has only five types as shown in Figure 1. For each output, we have all possibilities before performing next comparison, **match**(a_{i+1}, c_1):

$$\begin{cases} |A| \leftarrow |A| - 1, \text{ if } C = \emptyset \text{ OR } a_i(\sqsubset \text{ or } \equiv)\exists c_i \text{ OR } a_i \not\approx \forall c_i; \\ |A| \text{ remains, but } |C| \leftarrow |C| - 1, \text{ if } a_i(\oplus \text{ or } \sqsupset)\exists c_i. \end{cases}$$

Whenever $|A|$ remains, $|C|$ decreases by 1. If $|C|$ reaches to 0, then $|A|$ decreases by 1 in next iteration. Therefore, $|A|$ decreases monotonically, ensuring the termination of the algorithm. (q.e.d)

Lemma 2. *The upper bound of number of comparison in **s-self**(A) algorithm is $O(m^2)$, where $m = |A|$.* ■

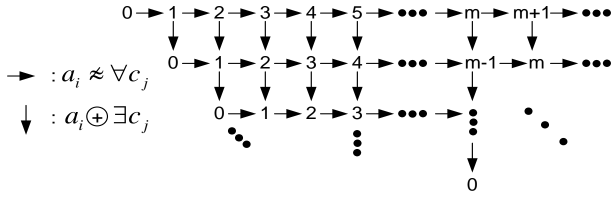


Figure 5: Structure of $|C|$ with respect to iteration path in s-self.

Scheme	Space	Time
p-CC	$(\frac{\max(m,n)}{p} + \min(m,n)) \times p$	$O((m+n)^2/p)$
p-DC	$(\frac{m}{p} + n) \times p$	$O((m+n)^2/p)$
p-DC$_{self}$	$(\frac{m}{p} + n) \times p$	$O((m+n)^2/p)$
p-DD1	$(\frac{\max(m,n)}{p} + \min(m,n)) \times p$	$O((m+n)^2/p)$
p-DD2	$(\frac{\max(m,n)}{p} + \min(m,n)) \times p$	$O((m+n)^2/p)$

Table 3: Summary of complexities ($m = |A|$, $n = |B|$, and $p =$# of processors).

PROOF. The lower bound of **s-self**(A) is simply $O(m)$ when all records are merged into a single record. In order to calculate the upper bound, since three match() relationships (i.e., $\sqsubseteq$, $\sqsupseteq$, and $\equiv$) always decrease the total number of records without the "re-feed", we will focus only on the other two relationships: $\oplus$ and $\not\approx$. Note that the maximum number of possible comparison (when a_i is compared to C) is bounded by $|C|$.

Let us draw a flow with respect to $|C|$ in Figure 5, where each number indicates $|C|$. The left-most node starts from 0, i.e. $C = \emptyset$, and $|C|$ increases by 1 without comparison, taking '$\rightarrow$', i.e., a_1 is added to C and becomes c_1. For the next record, a_2, if $c_1 \oplus a_2$, $|C|$ becomes 0, we draw '$\downarrow$'. If $a_2 \not\approx c_1$, taking '$\rightarrow$', $|C|$ becomes 2 and a_2 becomes c_2. In this manner, for a_3, the maximum number of comparisons is 2 ($= |C|$). We already prove in Lemma 1 that $|A|$ decreases when $C = \emptyset$ or $a_i \not\approx \forall c_j$, and $|A|$ remains the same when $a_i \oplus \exists c_j$. Therefore, $|A| = |A| - 1$ for '$\rightarrow$', and $|A|$ remains the same for '$\downarrow$'. At any moment, the number of $\rightarrow$ from the left-most node via one path is the same as $|A|$. Note that when $|A| = 0$, the algorithm stops. At any node of Figure 5, the number of '$\rightarrow$' is the same for any pathes from the left-most node. In addition, in the same column, the number of '$\rightarrow$' through any pathes is the same at any nodes. It is also true that the last node should be '$\rightarrow$'. If the last node is '$\downarrow$', then it indicates that the algorithm stopped at previous node. At any nodes, the number of comparison is the *sum* of nodes up to previous node through one path. Thus, the worst case is to stop at $|C| = 1$ or 2. For example, if the algorithm stops at node value 3 on the first row, the number of comparisons is 1+2=3. However, if it stops at node value 1 or 2 below, then we can add one more comparison with the same $|A|$. In addition, initial size of A is 3 since there are 3 '$\rightarrow$' when algorithm stops. In this manner, when $|A| = m$, the total number of comparisons in the worst case becomes $(1+2+...+m-2)+(m-1)+(m-2+...+1) = 2\sum_{i=1}^{m-2} + m - 1 = O(m^2)$. (q.e.d)

Lemma 3. *Space and time complexities of all six sequential algorithms are $O(m+n)$ and $O((m+n)^2)$, respectively.* ∎

Lemma 4. *All parallel linkage algorithms have polynomial upper bounds in time and space complexities.* ∎

PROOF. See Table 3. Details are omitted due to space constraint. (q.e.d)

6. EXPERIMENTAL VALIDATION

In this section, under various settings, we evaluate the performance of six sequential linkages (**s-CC**, **s-DC**, **s-DC**$_{self}$, **s-DD1**, **s-DD2**, and **s-DD3**) and five parallel linkages (**p-CC**, **p-DC**, **s-DC**$_{self}$, **p-DD1**, and **p-DD2**).

Two main metrics are used as baseline: *wall-clock running time*, denoted as **RT** and *number of comparison* for match(), denoted as **NC**. Then, the speedup and efficiency for parallel algorithms are defined in terms of RT and NC.

- **Speedup** shows the rate of increase for parallel system, and takes into account the overhead (e.g., time for startup, communication, synchronization for deadlock prevention, or data re-distribution) of parallel execution: $\textbf{speedup}_{RT} = \frac{RT_s}{RT_p}$, where RT_p and RT_s is the RT of the parallel and the "best" serial execution, respectively, and $\textbf{speedup}_{NC} = \frac{NC_s}{NC_p}$, where NC_p and NC_s is the NC of parallel and the "best" serial execution, respectively. Here NC_p is the accumulation of maximum NC among processors during iterations.

- **Efficiency** indicates the ability to gain proportionate increase in parallel speedup with the addition of more processors [9]: $\textbf{efficiency}_{RT} = \frac{\textbf{speedup}_{RT}}{\text{\# of processors}}$ and $\textbf{efficiency}_{NC} = \frac{\textbf{speedup}_{NC}}{\text{\# of processors}}$.

Errors are synthetically introduced to real citation data from DBLP according to two matching rates: (1) Internal Matching Rate of A: $\textbf{IMR}(A) = \frac{\text{\# of dirty records in } A}{\text{\# of all records in } A}$, and (2) Cross Matching Rate of A against B: $\textbf{CMR}(A,B) = \frac{\text{\# of records in } A \text{ that matches a record in } B}{\text{\# of all records in } A}$. By varying both IMR and CMR, we control the "dirty-ness" of data sets. When errors are introduced, four types of matching errors (e.g., $\equiv$, $\sqsubseteq$, $\sqsupseteq$, and $\oplus$) are *uniformly* distributed. For instance, to have an IMR of 0.4 for A, we synthetically generate 40% of A as matching (i.e., dirty) records, with 10% each for $\equiv$, $\sqsubseteq$, $\sqsupseteq$, and $\oplus$ types. To compare directly against P-Swoosh and P-Febrl and keep the experimentation manageable, data sets of 100 – 50,000 records in size are used. Despite their relatively small sizes, consistent performance patterns emerge (to be shown) and one can easily extrapolate the performance for very large data sets.

Distance metric between two citation records used Jaccard similarity with the threshold $\theta = 0.5$ by default. All proposed algorithms are implemented in the Distributed MATLAB, and executed in the LION-XO PC Cluster at Penn State[2], which includes 133 nodes, each with dual 2.4–2.6GHz AMD Opteron processor and 8GB–32GB memory. Since it is a multi-user multi-tasking machine, RT is measured as the average of multiple runs (i.e., 5-10).

6.1 Among Sequential Linkages

First, RT and NC among sequential algorithms are compared. Figure 6 shows both RT and NC of six sequential algorithms using IMR=0.0 and CMR=0.3 and 100 to 5,000

[2] http://gears.aset.psu.edu/hpc/systems/lionxo/

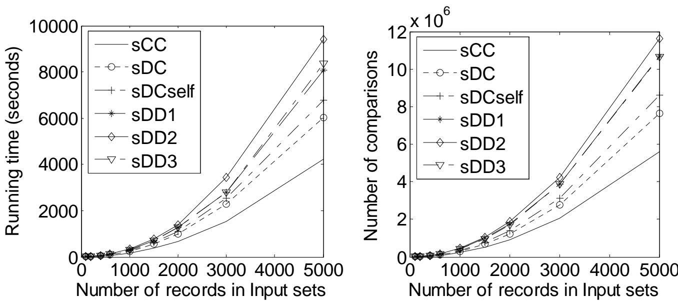
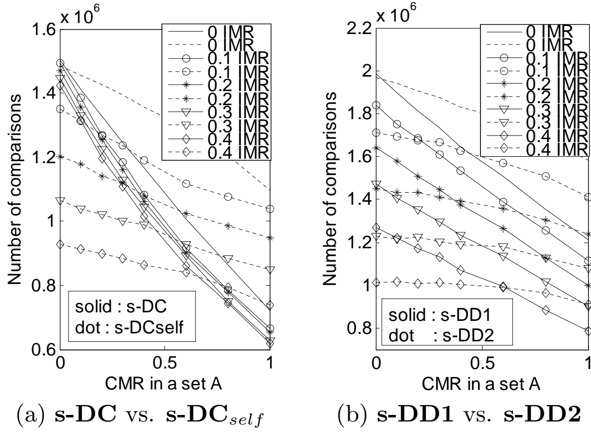


Figure 6: The RT and NC of six sequential algorithms (IMR=0.0 & CMR=0.3).



(a) **s-DC** vs. **s-DC**$_{self}$ (b) **s-DD1** vs. **s-DD2**

Figure 7: The total NC of four sequential algorithms with different IMR and CMR.

records. Although data is a clean-clean case (i.e., IMR=0.0), sequential algorithms for dirty-clean or dirty-dirty cases pretend *not* to know that they are clean so that comparison among all six algorithms is possible. Note that both graphs for RT and NC show consistent patterns of $O(N^2)$, where N is size of input. Since one does not need to compare records internally, **s-CC** shows the best RT and NT among sequential algorithms. For dirty-clean case, **s-DC** outperforms **s-DC**$_{self}$. Since **s-DC** compares records across A and B, due to high CMR of 0.3, after one iteration, **s-DC** matches and merge many records, reducing NC at subsequent iterations. For dirty-dirty case, RT of **s-DD1** and **s-DD3** are similar while both outperform **s-DD2**. Because of direct comparisons between A and B, records in A will be removed before being compared with records in the same set. Therefore, RT of **s-DD1** or **s-DD3** is faster than that of **s-DD2**.

Second, we compared how IMR or CMR affects the performance of sequential algorithms. We used five variations of IMR (0.0, 0.1, 0.2, 0.3, and 0.4) and eight variations of CMR (0.0, 0.1, 0.2, 0.3, 0.4, 0.6, 0.8, and 1) with 2,000 records. Both IMR and CMR are applied on A in dirty-clean case. For dirty-dirty case, both A and B take the same IMR, and CMR is applied only to A. Since results of both RT and NC are similar, here, we present only results of NC. In Figure 7, **s-DC**/**s-DD1** and **s-DC**$_{self}$/**s-DD2** are shown as solid and dotted lines, respectively. For dirty-clean case, NC decreases as both IMR and CMR increase in both **s-DC** and **s-DC**$_{self}$. Therefore, the data set with IMR=0.4 and CMR=1 (i.e., dirtiest data set) gives the best performance. The effect of IMR is more significant in **s-DC**$_{self}$ while the effect of CMR is more significant in **s-DC**.

This happens because, in **s-DC**$_{self}$, all redundant data are first merged during **s-self** stage, reducing CMR when **s-CC** is applied later. On the other hand, in **s-DC**, since the dirty set A is first compared to the clean set B, if CMR is high, then more records are merged at the first iteration. In conclusion, CMR (resp. IMR) is the dominant factor in **s-DC** (resp. **s-DC**$_{self}$). This conclusion can be interpreted as: the MR at the first iteration plays a major role on NC. Because of this, with IMR=0.0, **s-DC** always performs better. NC is significantly reduced by the increase of CMR on **s-DC**, i.e., **s-DC** performs better when it gets a higher CMR. As an example, with IMR=0.1, there is a cross-over point between **s-DC** and **s-DC**$_{self}$ at CMR=0.2. In general, **s-DC** is better with a higher CMR, and **s-DC**$_{self}$ is better with a higher IMR. The patterns of dirty-dirty case is analogous to those of dirty-clean case. With a large CMR and a small IMR, in general, **s-DD1** outperforms **s-DD2**.

Finally, Figure 8 illustrates the iterative nature of sequential linkage algorithm, **s-self**. Matched records at each iteration are merged into a new record, and re-compared to the rest of records at subsequent iteration. Therefore, sequential linkage algorithms continue until no more new merged record occur. With the data set of 2,000 records, depending on the default threshold θ for distance metric, Figure 8 shows that 3-8 iterations are needed to do complete self-clean.

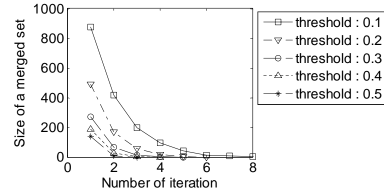


Figure 8: Iterations of the s-self algorithm.

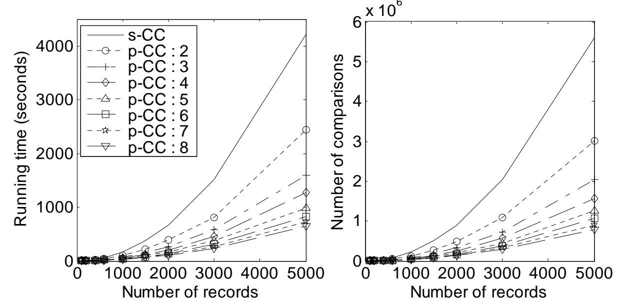


Figure 9: The RT and NC of p-CC (# in legend is # of processors).

6.2 Sequential vs. Parallel Linkages

Now, using **speedup**$_{RT}$ and **speedup**$_{NC}$, we compare sequential and parallel solutions. Up to 8 processors and 5,000 records are used. Since relative performance of all parallel algorithms, compared to sequential ones, are similar, we show only detailed behavior of **p-CC** under various characteristics.

In Figure 9, intuitively, both RT and NC decrease as # of

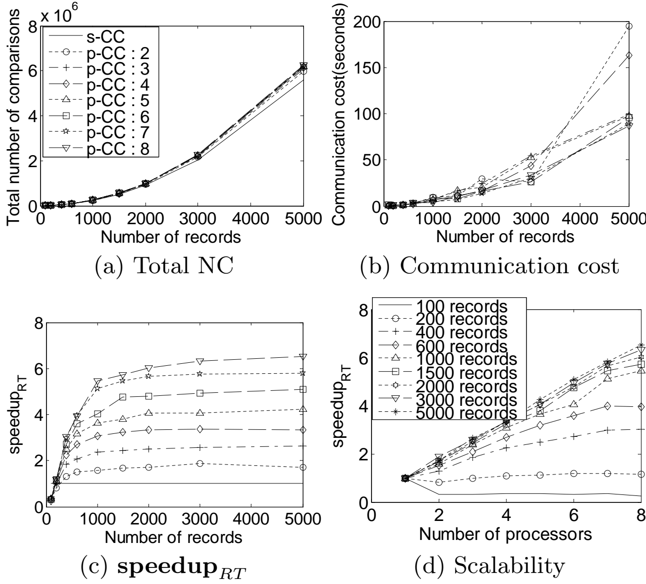


Figure 10: Details of p-CC (legend shown in (a)).

processors increases. Despite overhead cost of parallel execution, both RT and NC show the same pattern. However, speedup on RT is more affected by parallel overhead and it is shown in Figure 10. Figure 10(a) shows that although total # of NC increases as the input size increases, it gets little affected by # of processors used. That is, as more # of processors are used, it may increase involved overhead among processors (as communication cost shown in Figure 10(b)), but it does *not* increase # of comparison since **p-CC** has few redundant computations among processors – an ideal property for parallel algorithm. In our experimentation, communication overhead such as ones in Figure 10(b) typically consume 10% of total RT.

Therefore, in general, $\mathbf{speedup}_{RT}$ is less than $\mathbf{speedup}_{NC}$. Because it takes about 4-7 seconds to submit a parallel job and recollect final clean sets from processors, $\mathbf{speedup}_{RT}$ is very low when input data is less than 100 in Figure 10(c). With larger input data, however, RT is less affected by communication cost, and speedup with more processors is higher than that with less processors. When 8 processors are used, $\mathbf{speedup}_{RT}$ is close to 6.55. Finally, Figure 10(d) shows how scalable **p-CC** is. When data is sufficiently large and enough # of processors is used, **p-CC** shows linear increase of $\mathbf{speedup}_{RT}$ – another ideal property of parallel algorithm.

6.3 Among Parallel Linkages

Figure 11 shows the comparison of five parallel algorithms with respect to their efficiency. Also, both RT and NC are shown after being normalized (i.e., re-scaled to 0-1 range). Overall, $\mathbf{efficiency}_{NC}$ is better than $\mathbf{efficiency}_{RT}$ due to various overhead negatively affecting RT of parallel algorithms. Among parallel algorithms, **p-DD2** shows the best efficiency in both RT and NC. Because of the characteristic of input data (IMR=0.0 and CMR=0.3), using clean property gives better performance on both RT and NC. Thus, even though **p-DD2** gives the best efficiency overall, its RT and NC are also the highest. Specifically, note that RT of

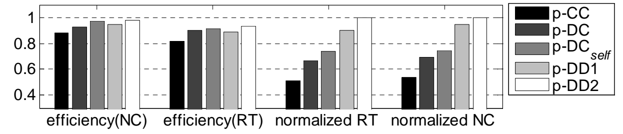


Figure 11: The RT and NC of five parallel algorithms (w. IMR=0.0, CMR=0.3, 8 processors, and 5,000 records).

s-DC is 1.11 times faster than that of $\mathbf{s\text{-}DC}_{self}$ and RT of **s-DD1** is also 1.11 times faster than that of **s-DD2**. In general, in terms of RT, parallel algorithms are in order of **p-CC** (fastest) <**p-DC**<$\mathbf{p\text{-}DC}_{self}$<**p-DD1**<**p-DD2** (slowest).

We also tried larger data set with more processors– 50,000 records with 16 processors and 32 processors. Since RT follows $O(N^2)$, an approximate equation for RT of a sequential linkage, say **s-CC**, can be obtained by *polyfit()* in MATLAB as: RT= $0.000173x^2 - 0.01035x + 3.645$, where x is the size of input data. That is, for 50,000 records, a sequential linkage algorithm such as **s-CC** would take about 120 hours to finish the job. However, when parallel linkage algorithms are used with up to 16 processors, RT can be reduced to about 9.17 hours with 16 processors and 4.7h with 32 processors. That is, we achieve $\mathbf{speedup}_{RT}$=13.08/$\mathbf{efficiency}_{RT}$=0.8175 and $\mathbf{speedup}_{RT}$=25.53/$\mathbf{efficiency}_{RT}$=0.7979 with 16 and 32 processors, respectively.

6.4 Against P-Swoosh and P-Febrl

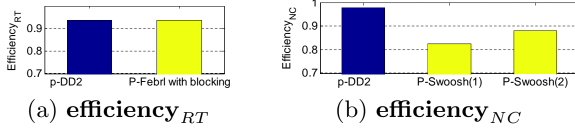
To our best knowledge, there are two parallel linkage solutions comparable to our proposals: P-Swoosh [13] from Stanford SERF project and parallel Febrl (P-Febrl) [5] from ANU record linkage project. Since space complexities of all three proposals are similar, let us focus on the comparison of speedup and efficiency. It is important to emphasize that in parallel experimentation, it is not straightforward to compare RT or NC directly. This is because RT may change depending on parallel execution model, choice of data characteristics, environment of execution system. However, the "ratio" of how much parallel solutions improve upon sequential solutions is meaningful. That is, if the ratio such as speedup in environment X is higher than that in Y, regardless of algorithmic details, one can argue that the parallel solution of X be superior to that of Y[3].

Since both P-Febrl and P-Swoosh studied only dirty-dirty case, here, we compare using **s-DD2** for sequential and **p-DD2** for parallel case. In addition, in [13], P-Swoosh reports only $\mathbf{speedup}_{NC}$ while in [5], P-Febrl reports only $\mathbf{speedup}_{RT}$. Therefore, we compare each against our solution separately. # of records and processors used in the experimentation are:

```
P-Febrl  : 20,000 records, 4 processors (w. blocking)
P-Swoosh : 5,000 records, 16 processors (NO blocking)
Ours     : 5,000 records, 8 processors (NO blocking)
```

Note that P-Febrl reports only results using *blocking* in linkage while both P-Swoosh and ours use nested-loop style linkage (thus no blocking). Therefore, the speedup of P-Febrl should be much higher than those of P-Swoosh and ours.

[3]The only matter that significantly affects the performance of both sequential and parallel algorithms is the characteristics of data sets. To ensure fair comparison, at all possible, we tried to compare similar input cases such as clean-clean or dirty-dirty.



(a) $\textbf{efficiency}_{RT}$ (b) $\textbf{efficiency}_{NC}$

Figure 12: Comparison with other parallel schemes.

As shown in Figure 12(a), however, our algorithms performs only 0.16% worse than P-Febrl (0.9360 vs. 0.9375). In [5], P-Febrl reports that their use of blocking and indexing on both sequential and parallel algorithms reduces substantial communication cost so that at the end only 0.35% of overall RT is due to the communication. In our experimentation, however, communication cost consumes about 7–13% of overall RT, leaving much room for improvement if blocking was used. Therefore, despite seemingly lower efficiency of our parallel algorithms in Figure 12(a) than that of P-Febrl, we believe that our parallel solutions is *more* efficient compared to P-Febrl. We plan to verify this claim in future work when blocking/indexing is combined with our parallel solutions.

As shown in Figure 12(b), against P-Swoosh, our algorithm, **P-DD2**, with $\textbf{efficiency}_{NC}$=0.9781 is 11.15% better than P-Swoosh (shown as P-Swoosh(2)) that has $\textbf{efficiency}_{NC}$ of 0.88. However, the result of P-Swoosh in [13] only considered the number of slave nodes in computing the efficiency, without including the master node. If the master node is also counted as # of processors (as it should be), then their $\textbf{efficiency}_{NC}$ drops to 0.825 (shown as P-Swoosh(1) in Figure 12(b)). Therefore, our proposals shows 11.15–18.56% improvement on $\textbf{efficiency}_{NC}$ against P-Swoosh[4].

6.5 Summary of Experimentation

With our input data set, RT and NC of sequential algorithms are ordered by **s-CC** (best) <**s-DC**<$\textbf{s-DC}_{self}$<**s-DD1**≈**s-DD3**< **s-DD2** (worst). Study on IMR and CMR shows that different algorithms can be used for better performance. Specially, for the one dirty set, **s-DD2** is always better than **s-DD1**. In parallel linkage, RT and NC of parallel algorithms are in the order of **p-CC** (best)<**p-DC**<$\textbf{p-DC}_{self}$ <**p-DD1**<**p-DD2** (worst). However, the order of $\textbf{efficiency}_{NC/RT}$ among parallel algorithms is **p-CC**<**p-DC**≈$\textbf{p-DC}_{self}$ ≈**p-DD1**<**p-DD2**. Efficiency of our algorithm performs comparably against P-Febrl (but ours do not use blocking and indexing while P-Febrl does) and performs better than P-Swoosh when the same comparison schemes are used on both parallel and sequential algorithms.

7. CONCLUSION AND FUTURE WORK

Parallel version of record linkage problem is studied in detail. For three input cases of clean-clean, dirty-clean, and dirty-dirty, we presented six sequential solutions and five parallel solutions. Our proposed parallel algorithms are shown to exhibit consistent improvement in speedup and efficiency when compared to sequential ones. In addition, compared to two other competing parallel solutions, ours show 11.15–18.56% improvement in efficiency.

[4]According to [13], their best $\textbf{speedup}_{NC}$ is more than 30 times using 10 processors. However, since they used radically different sequential and parallel schemes in so doing, this improvement of 30 is not meaningful. Therefore, we compare ours against their compatible model of FIX-1.

Many directions are ahead for future work. First, we plan to extend the current nested-loop style linkage solutions to support *blocking* as in blocked nested-loop style. This enables linkage solutions to quickly filter out unnecessary records so that only small number of candidate records are further examined. Second, we plan to do a large-scale validation using DBLP, CiteSeer, and ACM data sets (with 0.7-1 million citation records each).

Parallel linkage implementations and data sets that we used in this paper are available at:

`http://pike.psu.edu/download/cikm07/`

8. REFERENCES

[1] O. Benjelloun, H. Garcia-Molina, Q. Su, and J. Widom. “Swoosh: A Generic Approach to Entity Resolution”. Technical report, Stanford University, 2005.
[2] O. Benjelloun et al. “D-Swoosh: A Family of Algorithms for Generic, Distributed Entity Resolution”. Technical report, Stanford University, 2006.
[3] M. Bilenko, R. Mooney, W. Cohen, P. Ravikumar, and S. Fienberg. “Adaptive Name-Matching in Information Integration”. *IEEE Intelligent System*, 18(5):16–23, 2003.
[4] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. “Robust and Efficient Fuzzy Match for Online Data Cleaning”. In *ACM SIGMOD*, 2003.
[5] P. Christen, T. Churches, and M. Hegland. “A Parallel Open Source Data Linkage System”. In *Springer Lecture Notes in Artificial Intelligence*, Sydney, Australia, May 2004.
[6] R. Cilibrasi and P. M. B. Vitanyi. “The Google Similarity Distance”. *IEEE Trans. on Knowledge and Data Engineering (TKDE)*, (3):370–383, 2007.
[7] X. Dong, A. Y. Halevy, and J. Madhavan. “Reference Reconciliation in Complex Information Spaces”. In *ACM SIGMOD*, 2005.
[8] I. P. Fellegi and A. B. Sunter. “A Theory for Record Linkage”. *J. of the American Statistical Society*, 64:1183–1210, 1969.
[9] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *“Introduction to Parallel Computing (2nd Edition)”*. Addison Wesley, 2003.
[10] L. Gravano, P. G. Ipeirotis, N. Koudas, and D. Srivastava. “Text Joins in an RDBMS for Web Data Integration”. In *Int’l World Wide Web Conf. (WWW)*, 2003.
[11] M. A. Hernandez and S. J. Stolfo. “The Merge/Purge Problem for Large Databases”. In *ACM SIGMOD*, 1995.
[12] D. V. Kalashnikov, S. Mehrotra, and Z. Chen. “Exploiting Relationships for Domain-independent Data Cleaning”. In *SIAM Data Mining (SDM) Conf.*, 2005.
[13] H. Kawai et al. “P-Swoosh: Parallel Algorithm for Generic Entity Resolution”. Technical report, Stanford University, 2006.
[14] D. Menestrina, O. Benjelloun, and H. Garcia-Molina. “Generic Entity Resolution with Data Confidences”. In *VLDB CleanDB Workshop*, Seoul, Korea, Sep. 2006.
[15] B.-W. On, N. Koudas, D. Lee, and D. Srivastava. “Group Linkage”. In *IEEE ICDE*, Istanbul, Turkey, Apr. 2007.
[16] B.-W. On, D. Lee, J. Kang, and P. Mitra. “Comparative Study of Name Disambiguation Problem using a Scalable Blocking-based Framework”. In *ACM/IEEE Joint Conf. on Digital Libraries (JCDL)*, Jun. 2005.
[17] H. Pasula, B. Marthi, B. Milch, S. Russell, and I. Shpitser. “Identity Uncertainty and Citation Matching”. In *Advances in Neural Information Processing Systems*. MIT Press, 2003.
[18] S. Sarawagi and A. Bhamidipaty. “Interactive Deduplication using Active Learning”. In *ACM KDD*, 2002.
[19] D. A. Schneider and D. J. DeWitt. “A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment”. In *ACM SIGMOD*, Portland, OR, May 1989.
[20] W. Shen, X. Li, and A. Doan. “Constraint-Based Entity Matching”. In *AAAI*, 2005.
[21] W. E. Winkler. “The State of Record Linkage and Current Research Problems”. Technical report, US Bureau of the Census, Apr. 1999.