An Efficient Parallel Tree Mining Algorithm for Emerging Commodity Processors

Shirish Tatikonda
Department of Computer Science and Engineering,
The Ohio State University
tatikond@cse.ohio-state.edu

Srinivasan Parthasarathy
Department of Computer Science and Engineering,
The Ohio State University
srini@cse.ohio-state.edu

ABSTRACT

With the advent of multi-core technology, it is becoming increasingly evident that an effective parallel algorithm design is central to realize performance that is commensurate with this advancement. Additionally, along with effective parallelization strategies, algorithm designers must not only worry about memory access latencies but also memory bandwidth since technology constraints are likely to limit off-chip bandwidth to memory as one scales up the number of cores per chip.

As a step in this direction, from the perspective of data mining algorithm designs, in this article we consider the challenging problem of designing an efficient parallel algorithm for frequent tree mining. Our strawman solution (not a naive one) relies on a well known bijection between a labeled rooted tree and a sequence and leverages the properties of this bijection to realize an algorithm that is inherently array-based as opposed to tree-based that significantly improves the locality of said approach. We subsequently improve on this strawman by avoiding the maintenance of any state information during the mining process, and by employing a series of inter-linked pruning and computation chunking steps. We performed a detailed characterization of the employed optimizations on real data that demonstrated that our algorithm keeps a small working set, improves the memory locality, and alleviates the bandwidth pressure on the front side bus. For effective parallelization we rely on an adaptive load balancing strategy that leverages a combination of coarse-grained and fine-grained task and data partitioning. An empirical evaluation of our algorithm on modern day shared memory systems (SMPs) showed that our algorithm achieves a near linear speedup (up to 13 times on 16 processors) and reduces the memory footprint by up to 366 times without sacrificing on the run time.

1. INTRODUCTION

In recent years, spurred by the needs and demands of end applications, there has been a growing interest in the development of novel and efficient techniques for the mining and management of complex, structured and semi-structured data sets. For a large number of applications such data is often best represented or expressed in the form of labeled trees. Examples abound ranging from analysis and management of XML repositories [11, 19] to phylogenetic analysis [21], from Web mining [21] to analyzing movie documents [20], and from analyzing linguistic data expressed in terms of syntactic trees [10] to examining parse trees [2]. For such applications the essential problem may be abstracted to one of discovering frequent labeled subtrees from a database of rooted labeled trees [21]. This problem, often referred to as frequent subtree mining has been the subject of much recent research [1, 4, 9, 12, 13, 15, 16, 18, 21].

In this article we explore the design of an efficient algorithm for frequent subtree mining in the context of emerging architectures. The rationale for why we believe this to be an important problem to consider is as follows. First, mining frequent subtrees is often very time consuming. Optimizations that improve the efficiency of such algorithms are always desirable particularly when targeting problems of scale given that the knowledge discovery is inherently interactive and fast response times are thus desirable. Second, with the advent of multi-core processors it becomes imperative to identify scalable and efficient parallel algorithms to deliver performance commensurate with the number of cores. A fundamental challenge here is to ensure load balance in the presence of data skew. Third, a limiting constraint to efficient performance of data intensive applications on emerging architectures are the problems of memory latency and off-chip memory bandwidth (particularly for a large number of cores). Algorithms that are conscious of these performance limiting aspects are thus desirable.

To address these challenges in this work we propose a novel frequent subtree mining algorithm that improves the memory locality, shrinks the working set used, and reduces the off-chip traffic. We start with the algorithms TRIPS and TIDES, proposed as part of our previous research [15], which maintain embedding lists to trade off memory for speed – a viable option for current generation architectures. We improve on these algorithms by developing strategies which avoid the maintenance of such lists by dynamically constructing the relevant state, whenever needed. A series of linked optimizations, are then applied to reduce the computation and memory overhead. A careful characterization of the optimizations show that two of them, *tree pruning and computation chunking*, are particularly effective in improving the locality of the approach and to alleviate the bandwidth pressure – an important consideration for emerging multi-core architectures that are expected to be off-chip bandwidth bound if estimates by AMD and Intel are realized. Finally, we then present an intelligent, dynamic, and data-aware load-balancing strategy which enables one to adaptively modulate the type and granularity of the work being shared among processors enabling excellent speedups on up to 16 processing elements for real workloads.

We empirically show that our algorithm reduces the main memory usage of TRIPS by up to 366 times and improves the run time by up to 4 times. Through a detailed characterization study, we show that our optimizations systematically improves individual steps of the algorithm. Furthermore, we show that, on two real data sets, our algorithm keeps small working set sizes of approximately 8-16 KB. Evaluation of our load balancing strategies show up to 13-fold speedup on 16 processors.

Specifically, we make the following contributions in this article: **First**, we propose a new algorithm that dynamically constructs the embedding list by intelligently operating on the sequence representations. **Second**, we propose a novel tree mining algorithm that completely eliminates the need for the embedding lists by leveraging a series of inter-linked optimizations. **Third**, we empirically show that our algorithm is suitable for emerging chip multi-processor systems by performing a detailed characterization study of our algorithms. **Finally**, we propose different parallel algorithms by designing various coarse-grained and fine-grained task and data partitioning strategies.

2. BACKGROUND AND RELATED WORK

We assume that the reader has an understanding of the basic concepts in graph theory [6]. An *induced* subtree of a tree preserves the parent-child relationships and an *embedded* subtree preserves the ancestor-descendant relationships. Unless otherwise stated, a *tree* refers to a rooted, ordered, labeled tree and a *subtree* or a *pattern* refers to an embedded subtree.

Problem Definition: Given a database of trees D and a user-defined minimum support threshold *minsup*, the goal of frequent subtree mining is to enumerate all the subtrees, which occur as the embedded subtrees in more than *minsup* trees in the database.

The first tree mining algorithm, *TreeMiner*, was proposed by Zaki [21]. A limitation of this method is that it uses pointer-based dynamic data structures and uses a lot of memory as we demonstrate in [15]. Chi *et al* [5] present an excellent overview of tree mining. Wang *et al* proposed two algorithms, *Chopper* and *XSpanner* [18] where the former recasts subtree mining into sequence mining and the latter generates frequent patterns without *explicit* candidate generation. A potential problem with this approach is that the recursive projection may again lead to a lot of pointer chasing and poor cache behavior [15]. There exists several other algorithms which slightly differ in the type of trees which are mined such as free, unordered, closed, maximal subtrees [1, 4, 9, 12, 16] or in the type of support definition that they use such as transaction-based support [13]. As part of previous research, we have developed the adaptive task partitioning strategies for frequent subgraph mining [3]. We have done a characterization study of the frequent itemset mining algorithms and pointed out that most of the algorithms grossly under-utilize a modern processor [7]. In this work, we focus on developing new memory-aware algorithms for frequent subtree mining and we show their applicability for emerging architectures.

2.1 TRIPS and TIDES

We have proposed two algorithms, TRIPS and TIDES for mining frequent subtrees [15]. They transform the data set trees into sequences and operate on them. A number of tree sequencing methods have been proposed and used in the context of tree mining [18, 21] and tree indexing [11, 19, 15]. We deal with two types of sequences, *prüfer* sequences and *depth-first* sequences.

Representation: We represent each tree with a Consolidated Prüfer Sequence (CPS) that consists of two other sequences, NPS – Numbered Prüfer Sequence and LS – Label Sequence. For simplicity, we refer to CPS as the prüfer sequence. They are based on the post-order traversal (PO) of the tree. While constructing NPS, at each step, the leaf with smallest PO number is removed and the PO number of its parent is appended to the partially constructed NPS. The LS is constructed using the labels of the leaf nodes, which were deleted at each step [15]. Example trees along with their *prufer* sequences are shown in the figure 1. Against each node, the label and the PO number is shown. Depth first sequences are constructed in a similar way to CPS except that they are based on depth-first traversal of the tree as opposed to the post-order traversal. Both prüfer and depth-first sequences *provide a bijection* between rooted ordered trees with n nodes and the sequences of length n.

TRIPS relies on Prüfer sequence encoding whereas TIDES is based on depth-first sequences. These algorithms are generic in the type of subtrees which are mined – induced, embedded, labeled, unlabeled, ordered, unordered, or edge-labeled. They employ a pattern growth approach for systematically generating the candidate patterns. This approach starts with a seed pattern and the pattern is recursively grown edge by edge in order to generate new candidate patterns. The seed patterns along with the subtrees grown from it often referred to as *equivalence classes.* By transforming the operations on the trees to operations on the sequences, these algorithms are made to work on simple array-based structures rather than pointer-based tree structures. For example, a simple sequence extension mimics the operation of growing a tree with an edge. Such a candidate generation process produces every possible candidate subtree (i.e., complete) and produces and operates on each candidate only once (i.e., non-redundant). This efficient candidate generation process makes the support counting step to be a trivial task of count accumulation.

We outline the essence of TRIPS and TIDES in the algorithm 1. A set of frequent labels $F1$ is first constructed while reading the data set. For each frequent label $f \in F1$, $mineTrees$ is invoked to recursively enumerate the frequent subtrees in f's equivalence class.

The procedure $mineTrees$ is invoked with a pattern pat, an extension point e, and a list of trees in which pat occurs as a subtree. An *extension point* that is defined with respect to a pattern P, denoted as a pair (*lab*,*pos*), represents a subtree that is obtained by attaching a vertex with label *lab* to a vertex in P whose post-order number is *pos*. $mineTrees$ first constructs a list of trees *newtidlist* in which *newpat* has at least one subtree isomorphism and an embedding list $EMList$ that stores the exact location of each isomorphism

Algorithm 1 TRIPS and TIDES

```
Input: $D = \{T_1, T_2, \ldots, T_N\}, minsup$
  $F_1$ = readTrees ($D$)
  for each $f$ in $F_1$
     mineTrees ($NULL$, ($f$, $-1$), $D$)

mineTrees ($pat$, extension $e$, tidlist)
 1: $newpat$ = extend ($pat$, $e$)
 2: output $newpat$
 3: ($newtidlist$,$EMList$) = findMatches($newpat$, tidlist)
 4: $H$ = NULL
 5: for each $T$ in $newtidlist$
 6:    for each node $v$ in $T$
 7:       for each match $m$ in $EMList[T]$
 8:          if $v$ passes connectivity check against $m$ then
 9:             add the resulting extension to $H$
10: for each $ext$ in $H$
11:    if $ext$ is frequent then
12:       mineTrees ($newpat$, $ext$, $newtidlist$)
```

(line 3). For efficiency reasons, $EMList$ is carried across the recursive calls and appended with new entries, which are removed when the call returns [15]. Each node v in T is processed against each of the $newpat$'s matches m to generate the extensions (lines 5-9). A *connectivity check* is performed to see if v is a valid extension to $newpat$ against m. This check passes if v or any of its ancestors is attached to a node that is on the left most path of m. Note that while mining induced subtrees, the check is *not* evaluated for ancestors but only for the parent. Resulting extension points along with their supports are stored in a hash structure H (line 9). Finally, each frequent extension in H that represents a new frequent subtree is processed recursively by invoking $mineTrees$ (lines 10-12).

Throughout this paper, we evaluate all the proposed algorithms *only against TRIPS*. Note that this strawman is not a naive one and in fact it is shown to outperform TreeMiner, XSpanner, and Chopper [15] and therefore we *do not make any comparisons* against these algorithms. In [15], we showed that the performance of TIDES is similar to TRIPS and the algorithms and optimizations proposed for TRIPS can also be applied to TIDES.

Limitation: Though embedding lists usually reduce the overall mining time, they can potentially grow arbitrarily in size and can end up degrading the performance. Consider a tree that is a path of length n where every node of the path has the same label, say A. If the pattern of interest has a single node, there will be exactly $\binom{n}{1}$=n entries in the embedding list. Suppose the pattern is an edge $A - A$, then the embedding list would contain $\binom{n}{1} + \binom{n}{2} = \frac{n(n+1)}{2}$ entries. Recall that the list is carried across the recursive calls and appended with entries. Similarly if the pattern is the full path of n nodes, the number of entries in the list would be $\sum_{i=1}^{n} \binom{n}{i} = 2^n - 1$, even though there is only a single match for that pattern. Therefore the embedding list can potentially grow exponentially in size.

Such cases often arise in real world data sets. For example, the number of matches of a particular 3-node pattern in *Cslogs* data set (see Section 3.7), when grown with an edge, was increased from $141,574$ to $2,337,127$ amounting to an increase in the embedding list size from 1.2 MB to 19.02

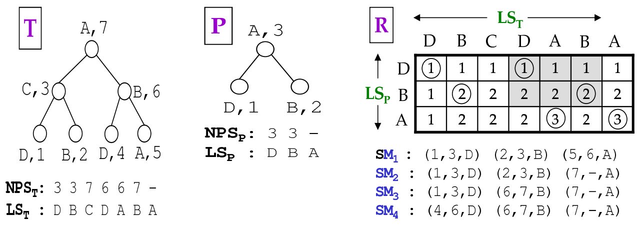


Figure 1: Example Tree, Pattern, and R-Matrix

MB. Such huge embedding lists increase the memory footprint and can potentially result in a number of cache misses and page faults. Furthermore, embedding lists complicates the design of parallel algorithms as sharing them among different processors is not trivial. These issues motivated us to explore the use of mining algorithms which do not make use of static, recursively-grown embedding lists.

3. SEQUENTIAL OPTIMIZATIONS

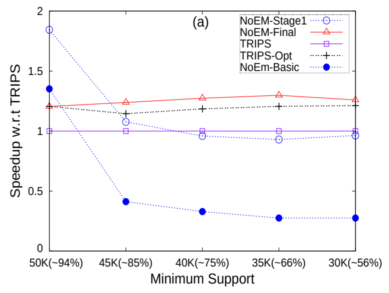
In this section, we describe a series of optimizations by which the strawman algorithm 1 can be improved. We then present a detailed characterization study that shows that our algorithm systematically reduces the overhead incurred at various phases of the mining process and exhibits excellent cache performance. Furthermore, we empirically show that our algorithm keeps small working sets and reduces the off-chip traffic making them a viable option for systems with small caches such as modern day CMP architectures.

3.1 Tree Pruning and Recoding (PRUNE)

Tree pruning and encoding draws inspiration from the field of itemset mining. Many itemset mining algorithms often prune the transactions involving infrequent items and recodes them [7, 8]. This is an idea that has not been well exploited in the context of tree mining and we leverage it here. We expect that such an approach will decrease the amount of resident memory and also improves the performance by partially reducing potential page faults and cache misses. Unlike the traditional set-like transactions, trees have an inherent structure that makes tree pruning a non-trivial task. We apply the algorithm 2 in order to prune and recode the database trees.

The intuition behind this algorithm is that every infrequent node, except the root node, in the tree is removed and its child nodes are attached to its nearest ancestor that is not pruned away. The pruning step alters the number of nodes and the recoding step alters the labels. Therefore, this strategy changes both the label sequence and the numbered prüfer sequence of T. The number of nodes in the recoded tree is found in lines 4-6. The recoding process is carried out from root to leaves. Therefore, the prüfer sequence is scanned from right to left (line 8). An array *mapping* is maintained to keep track of the new post-order traversal numbers. The root node is added to the pruned tree $T_{recoded}$ with its recoded label. If v is a frequent node other than the root then, in lines 11-13, it is added with its recoded label and the *new* post-order number of its parent. Note that if that parent in T is pruned away, the post-order number would refer to the nearest ancestor the deleted parent that is frequent. In the worst case, a leaf node can become a child of the root node if all the nodes in between are infrequent.

Evaluation: The effectiveness of this optimization evalu-



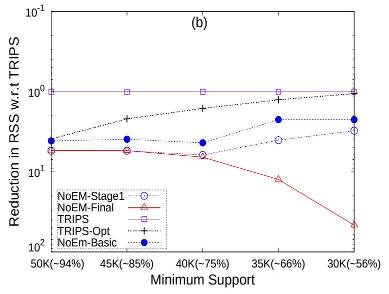


Figure 2: Treebank: Performance comparison with TRIPS as the baseline (a) Mining time (b) Memory

Algorithm 2 Tree pruning and recoding algorithm

```
Input: Database of trees, $D$
Input: Recoded labels of frequent nodes, $newlabels$
 1: for each tree $T$ in $D$
 2:   $n \leftarrow$ size of $T$
 3:   $T_{recoded} \leftarrow null$
 4:   for each node $v \in T$
 5:     if $v$ is the root or $v$ is frequent then
 6:       increment $count$
 7:   $ind \leftarrow count - 1$
 8:   for $i$ from $n$ to 1
 9:     $v \leftarrow i^{th}$ node in the prüfer sequence of $T$
10:     if $v$ is root then
11:       add $v$ to $newlabels$, if not present
12:       add $v$ to $T_{recoded}$ with recoded label
13:       $mapping[n] \leftarrow count$
14:     else if $v$ is frequent then
15:       $u \leftarrow (newlabels(v.label), mapping[v.parent])$
16:       add $u$ to $T_{recoded}$
17:       $mapping[i] \leftarrow ind$
18:       decrement $ind$
19:     else
20:       $mapping[i] \leftarrow mapping[v.parent]$
```

ated on Treebank data set (TB) [1] is shown in the figure 2 as "TRIPS-Opt". This data set, derived from computational linguistics domain, has a total of 52,851 trees with more than 1.3 million distinct labels [15]. In figure 2a, we show the speedup achieved by other algorithms considering TRIPS as the baseline and the reduction in memory footprint is shown in the figure 2b. We approximate the memory footprint size of an algorithm to be its resident set size (RSS). Note that the Y-axis in figure 2b is in *reverse* direction to indicate the reduction.

Though "TRIPS-Opt" exhibited a better run time performance over TRIPS, its memory footprint did not improve as much. With lower support levels, the memory usage is so high (e.g. 1.5 GB at $30K$ [2]) that the relatively small benefits from pruning are overshadowed. At even low supports, this increased memory usage can potentially slow down the mining process. For the $TB - 40K$ experiment (TB is the data set and $40K$ is *minsup*), tree pruning and recoding has reduced the size of the data set representation from 27 MB to 11.5 MB. It improved the mining time of TRIPS from 273 seconds to 230 seconds while reducing the memory footprint from 222 MB to 138 MB. Though this improvement is marginal, as we show later this optimization buys us a lot when used in conjunction with other optimizations.

3.2 On-the-fly Embedding Lists

In this section, we propose an algorithm that dynamically constructs the embedding list, uses it, and then destroys it. Such a dynamic approach enables the use of optimizations like computation chunking (see Section 3.7) and limits the memory usage of the algorithm. Specifically, the problem of on-the-fly embedding list construction is to find the set of all subtree isomorphisms or matches for a given pattern P in a given tree T (line 3 in Algorithm 1). Without loss of generality, assume that both P and T are represented as prüfer sequences (see Section 2.1) and the number of nodes in P and T are m and n, respectively.

Theorem 3.1. *If a pattern P is a subtree of a tree T, then the label sequence of the pattern (LS_P) is a subsequence of the label sequence of the tree (LS_T).*

Proof. The intuition is that if P is a subtree of T, then the order among the sibling nodes of P is preserved from T. Therefore, while constructing the prüfer sequence for P, the order in which nodes in P are deleted would be the same as the order in which corresponding nodes in T are deleted. □

Property 3.1. *The label sequence of the pattern LS_P is a subsequence of the label sequence of the tree LS_T if and only if LS_P is the longest common subsequence of LS_P and LS_T.*

Algorithm 3 outlines the procedure in which on-the-fly embedding lists are constructed. This algorithm is inspired by some of the recent results we obtained in the context of XML indexing [14]. It recasts the subtree isomorphism problem into the subsequence matching problem. From theorem 3.1 and the property 3.1, we derive that if P is a subtree of T, then the longest common subsequence (LCS) of their label sequences should be LS_P. Once it is known that P is indeed a subtree, its matches are found by enumerating all subsequence matches of LS_P in LS_T. Note that most of the classic LCS algorithms find only the length of LCS. Since Theorem 3.1 gives only the necessary condition, a post-processing step is employed to filter out the false positive matches.

Step 1 - *Computation of LCS length*: In this step, we check whether LS_P is a subsequence of LS_T, by computing the length of their LCS. We do so by constructing the R-matrix using the equation 1 [17]. If $R[m,n]$, the LCS length, is different from m, we can conclude that P is not a subtree

[1] `http://www.cs.washington.edu/research/xmldatasets/`
Experiments are conducted on a 900 MHz Intel Itanium 2 dual processor system with 12 GB of main memory.
[2] We chose high supports for TB as the data set is highly associative.

of T (from Property 3.1). Example pattern, tree, and the corresponding R-Matrix is shown in the figure 1.

$$R[i,j] = \begin{cases} 0, & if\ i=0, j=0 \\ R[i-1,j-1]+1, & if\ LS_P[i] = LS_T[j] \\ max(R[i-1,j], R[i,j-1]), & if\ LS_P[i] \neq LS_T[j] \end{cases} \tag{1}$$

Step 2 - *Subsequence Matching*: Given that LS_P is a subsequence of LS_T (from previous step), we now find the set of all subsequence matches of LS_P. The subsequence match is constructed by backtracking on the R-Matrix starting from the bottom-right entry (lines 4-12 of the algorithm 3). Therefore, the match is constructed from right-to-left, in the reverse order of LS_P. Note that when the labels in P and T match in line 4, the match location is noted in *subiso* (line 5) and the subsequence match length L is incremented (line 6). Define SM to be the resulting subsequence match that is constructed from entries of NPS_T and LS_T whose positions are given by *subiso*. Specifically,

$$SM = ((i_1, NPS_T[i_1], LS_T[i_1])...(i_m, NPS_T[i_m], LS_T[i_m]))$$

where i_j is a PO number in T and i_j = *subiso*$[j]$. In the figure 1, there are 4 subsequence matches, each of which is shown as SMs.

Algorithm 3 On-the-fly embedding list construction

```
Input: $P = (LS_P, NPS_P)$, $T = (LS_T, NPS_T)$
  $R \leftarrow$ computeLcsMatrix($LS_P$, $LS_T$);
  say $m \leftarrow |LS_P|$, $n \leftarrow |LS_T|$
  if $R[m][n]$ != $m$ then
    return
  processRMatrix ($m$, $n$, 0)

processRMatrix ($pi$, $tj$, $L$)
 1: if $L = m$ then
 2:   if $subiso[..]$ corresponds to a subtree then
 3:     update $EMList[T]$ with $subiso$
 4: if $LS_P[pi] = LS_T[tj]$ then
 5:   $subiso[m-L] \leftarrow tj$
 6:   processRMatrix ($pi-1$, $tj-1$, $L+1$)
 7:   processRMatrix ($pi-1$, $tj$, $L$) // Not required
 8:   processRMatrix ($pi$, $tj-1$, $L$)
 9: else if $R[pi-1, tj] > R[pi, tj-1]$ then
10:   processRMatrix ($pi-1$, $tj$, $L$)
11: else
12:   processRMatrix ($pi$, $tj-1$, $L$)
```

Step 3 - *Structure Matching*:

We prune the false positive matches resulting from the previous step by matching the structure formed by the subsequence match SM with the structure of P. To establish such a match, our algorithm needs to make only a *single pass* over SM.

Similar to the subsequence matching step, we establish the structure match from right-to-left i.e., from root to leaves. This reverse order ensures that a node in SM is mapped to a node in T *only after* all of its ancestor nodes including its parent are mapped. We perform a *structure agreement check* to match each node in the subsequence match. However, the root nodes i.e., the right most entries are matched without checking. This check translates the parent-child relation in P into an ancestor-descendant relation in SM i.e., in T. More formally, SM and P are said to agree on structure at position k *iff* $LS_P[k]$=$LS_T[i_k]$ and $NPS_P[k]$ in P is mapped to an ancestor of $NPS_T[i_k]$ in T [3]. Note that $NPS_P[k]$ is mapped before the node at position k because the structure match is established from root to leaf. Once the structure match is established, the embedding list is updated with the locations of the match as given by *subiso* (line 3). Please refer to [14] for a formal description of the structure matching algorithm.

For the example in the figure 1, there are only 2 structure matches, SM_2 and SM_3. For SM_1, the root in P is mapped to node 6 in T. At position k=2 (i_k=2), the parent node $NPS_P[2]$=3, whose mapped node is 6. However, $NPS_T[i_2]$=3 is not an ancestor of 6 resulting in a structural mismatch. Similarly, SM_4 fails the structure agreement check at position 1.

Evaluation: Performance comparison between the algorithm 3, TRIPS, and its optimized version is shown in the figure 2. On-the-fly embedding lists, labeled as "NoEm-Basic", clearly trades off the computation for memory usage. Expensive embedded subtree isomorphism checks in *find-Matches* slow down the mining process. However, dynamic embedding lists result in a significant improvement in terms of memory usage. For the running example of $TB-40K$ experiment, this algorithm shown a 3.6-time slow down in mining time and a 2.7-fold improvement in memory footprint size from 138 MB to 51 MB. Though the amount of reduction in memory usage is encouraging, the slowdown in running time is unsatisfactory.

3.3 Label Filtering (LF)

Poor performance of the algorithm 3 is partly due to a high number of recursive calls to *processRMatrix*. Consider the fact that a total of 10 billion calls are made, while mining $TB-40K$, to find only 413 million subsequence matches. Consider the matrix entries $R[*,k]$, where $LS_T[k] \notin LS_P$. These entries simply carry forward the LCS values from $R[*,k-1]$ to $R[*,k+1]$. The recursive calls made on $R[*,k]$ (lines 9-12 in algorithm 3) do not add to the subsequence match and hence can safely be removed. Before constructing the R-matrix, LF prunes the label sequence of T from the labels which are not present in P. In the figure 1, the third column corresponding to C can thus be removed. It not only *decreases the number of recursions* but also *shrinks the R-matrix making it fit in few cache lines.*

LF reduced the number of recursions in $TB-40K$ experiment by 4.5 times to 2.2 billion thereby improving the mining time.

3.4 Dominant Match Processing (DOM)

The effectiveness of label filtering completely depends on the distribution of labels in P over the nodes of T. If every label in T is in P, then LF has no effect on the performance. In this optimization, we further reduce the number of recursions by limiting the recursion to only a selected few cells in the R-matrix.

Say that $R[i,j]$ and $R[k,l]$ are two cells at which the LCS length is incremented. Assume also that $\nexists(x,y)$ such that $i<x<k$, $j<y<l$, and $LS_P[x] = LS_T[y]$. All the cells between $R[i,j]$ and $R[k,l]$ carry the LCS length from $R[i,j]$ to $R[k,l]$ and do not contribute to the subsequence match. We

[3]For induced subtrees, $NPS_P[k]$ should be mapped to the parent node of $NPS_T[i_k]$.

refer to the cells like $R[i,j]$ and $R[k,l]$ as *dominant matches*. Backtracking from $R[k,l]$ can directly jump to $R[i,j]$ *without* going through all the intermediate cells. In the figure 1, dominant matches are encircled. For example, $R[1,4]$ and $R[2,6]$ are dominant and all the other shaded cells simply carry the LCS length from $R[1,4]$ to $R[2,6]$. This *significantly reduces the total number of recursions* made on the R-matrix.

Both LF and DOM jointly reduced the number of recursions in the $TB-40K$ experiment to 554 million, compared to 10 billion by the algorithm 3, that is closer to the number of subsequence matches, 413 million.

3.5 Simultaneous Subsequence and Subtree Matching (SIMUL)

Both LF and DOM target the subsequence matching step in Algorithm 3 and they do not address the issue of false positive subsequence matches, which are fed into the structure matching phase. While mining for $TB-40K$, approximately 7 out 10 subsequence matches were false positives, in fact, 124 million out of 413 million.

We now design an algorithm that *completely eliminates the false positives.* This algorithm detects them in the early stages of subsequence matching, instead of waiting until the complete match is found, saving the futile time spent in processing these matches. It establishes the structure matching, by performing structure agreement checks, as the subsequence match is constructed. Such a unified approach can be realized using our sequence-based algorithms because of the two fundamental reasons. *first*, both the subsequence and the structure matching steps process the nodes in P in the same direction, right-to-left. *second*, a structure agreement check at a node needs only its ancestors, for which the structure match is already established. Complete details of the unified algorithm can be found in [14].

Evaluation: Trend line "NoEm-Stage1" shows the performance of the algorithm 3 optimized with all the strategies presented so far including tree pruning and recoding. When compared to the basic algorithm "NoEm-Basic", it gave up to an average speedup of 3 times with a marginal improvement in memory usage. In particular for the $TB-40K$ experiment, the unified algorithm with all the other optimizations gave a 4-fold run time improvement over the basic algorithm. More importantly, it showed *only* 23% *slowdown* when compared to optimized TRIPS (see Section 3.1) while *reducing the memory footprint by* 3.8 *times.*

3.6 Loop Inversion (LOOP)

Recall that the extensions are found by processing the nodes in tree T with respect to all the matches of P (lines 6-7 of the algorithm 1). Such repeated scans over the entire large embedding list may hinder the performance. If we invert the loops in lines 6-7, we hope that the tree would sit in the cache while we process the nodes against every match. However, this strategy exhibited a degraded performance with both the algorithms 1 and 3. Note that the number of scans made on T with the loop inversion is equal to the number of matches of P in T. If T is large enough that a single scan on T incur few cache misses, then the repeated scans on the tree would result in a poor cache performance. Though loop inversion did not give any performance improvement it worked as a stepping stone in our way to design the next optimization.

3.7 Computation Chunking (CHUNK)

Algorithm 4 Fully optimized tree mining algorithm

```
mineTrees ($pat$, extension $e$, tidlist)
  for each $T$ in tidlist
    construct $R$-Matrix for $T$ and $newpat$
    processRMatrix ($m$, $n$, $m$)
  for each $ext$ in $H$
    mine ($newpat$, $ext$) recursively
processRMatrix ($pi$, $tj$, $L$)
 1: if $L = 0$ then
 2:   add $SM$ to $EMList$ and add $T$ to $newtidlist$
 3:   if $|EMList|$ % $10 = 0$ then
 4:     for each match $m$ in $EMList$
 5:       for each node $v$ in $T$
 6:         if $v$ passes connectivity check against $m$ then
 7:           add th4e resulting extension to $H$
 8:   $EMList \leftarrow null$
 9:   return
10: for $k = tj$ to 1
11:   if $R[pi][k]$ is dominant & $R[pi][k]{=}L$ then
12:     $SM[k] \leftarrow (LS_T[tj], NPS_T[tj])$
13:     if agreeOnStructure ($P$, $SM$, $k$) then
14:       processRMatrix ($pi-1$, $tj-1$, $L-1$)
```

The patterns with a very large number of matches (see Section 3.7) result in huge dynamic lists, defeating the purpose of creating them. We now design an optimization, computation chunking, that *eliminates the embedding lists entirely.* Recall that with the loops in lines 6-7 of the algorithm 1 inverted, the tree T is scanned for each match m to find the extensions. We note that this can be performed at the time when m is discovered in $findMatches$. In other words, as soon as a match is generated, a scan on the tree can be made eliminating the need for it to be stored. But as we learned in the section 3.6, a match-by-match processing is not very effective. We therefore group a set of matches as one *chunk* and we process the nodes in T once for each chunk.

Though the underlying principle behind chunking is similar to *tiling*, they are quite different. *First*, tiling groups a set of data items and performs a computation on that set whereas chunking groups a set of computations and applies that set on to a single data item (here, T). We thus refer to these chunks as *computation chunks. Second*, tiling improves the cache performance by grouping the nodes of T into parts such that each individual part fits in the cache. In contrast, computation chunking reduces the number of cache misses by reducing the number of scans on T (without dividing T). Therefore, the algorithm 3 with computation chunking, unlike tiling, *does not depend on any hardware parameters* such as cache size or line size making it a *cache oblivious* algorithm as opposed to a cache-conscious algorithm. For our studies, we set the chunk size to have 10 matches as we did not observe a significant improvement with larger chunks. Please note that the computation chunking *can not be applied* to TRIPS or "TRIPS-Opt" due to embedding lists.

Complete details of the chunked version is shown in Algorithm 4. The matches are grouped into chunks in line 2 and the tree is scanned for each chunk to generate the extensions in lines 3-7. Lines 10-14 recurse on the R-Matrix to construct the subsequence match SM. This algorithm does

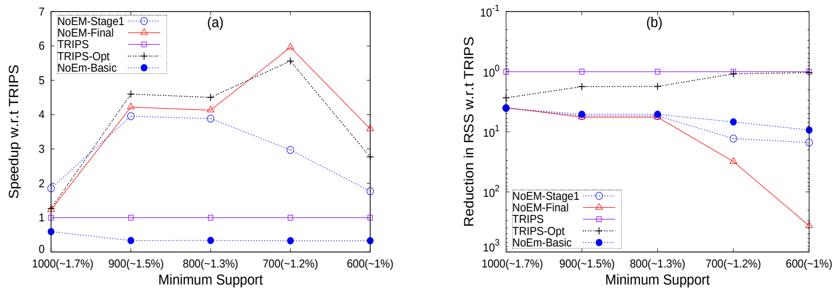


Figure 3: Performance comparison on Cslogs with TRIPS as the baseline (a) Mining time (b) Memory

no additional recursive invocations unlike lines 9-12 in Algorithm 3. Such chunk-level processing result in an extremely good cache performance (see Section 3.8).

Evaluation: The performance of our fully optimized algorithm, labeled "NoEm-Final", is shown in the figure 2. When compared to TRIPS and its optimized variant, our algorithm gave up to a 45-fold reduction in the memory footprint size, from 1.5 GB to 34 MB for $TB-30K$. As a result, the mining time has reduced from TRIPS' $3,660$ seconds to $2,900$ seconds. Notably, it is also faster than the optimized TRIPS. Specifically for $TB-40K$, "NoEm-Final" exhibited a 7-fold speedup over "NoEm-Basic" and when compared to TRIPS and its optimized version, it reduced the memory usage by 6.5 and 4 times, respectively.

Results on Cslogs: Figure 3 compares the effectiveness of all the algorithms on Cslogs data set (CS) [4]. This data set contains $59,691$ trees with $13,361$ distinct labels. The basic algorithm 3 again exhibits the trade-off between the run time and the memory usage. Subtree matching optimizations (LF, DOM, SIMUL) along with pruning and recoding bump up the run time performance with a marginal reduction in memory usage (trend "NoEm-Stage1"). Similar to the results on Treebank, the algorithm 4 demonstrates a better run time performance and memory usage when compared to "TRIPS" and even "TRIPS-Opt". Specifically at $minsup$=600, it reduced the resident memory size by 366-*times* from 3.8 GB to 10.7 MB with a 3.7-fold speedup in run time. The two orders of magnitude reduction is due to a 6-node pattern that has approximately 474 million matches.

3.8 Characterization study for CMP architectures

In this section, we study how the optimizations we have presented affect cache performance at different levels (L1, L2 and L3), to what degree one is able to limit the working set size, and how they affect the bandwidth pressure on the front side bus. Through this detailed characterization study we hope to show that our algorithms are viable choices for emerging chip multi-processor (CMP) systems. We used the $PAPI$ library for performance instrumentation[5].

3.8.1 Analysis of cache performance

In Figure 4a, we considered the number of cache misses of the basic algorithm 3 as the baseline and measured the reduction in misses as we add each optimization. We used Treebank data set with $minsup$=$45K$ for this experiment [6]. We observed the similar results on Cslogs data set. Each trend line is labeled as $C-P$, where C denotes the cache (L1,L2, or L3) and P represents the number of processors (1 or 2) on which the algorithm is run. We parallelized the algorithm 4 using various coarse-grained and fine-grained data and task partitioning strategies (see Section 4).

SIMUL reduces the number of accesses to the database trees by eliminating the false positive subsequence matches and PRUNE shrinks the trees by pruning infrequent nodes. The computation is thus localized to caches thereby reducing the total amount of off-chip traffic. Both SIMUL and PRUNE are thus targeted at L2 and L3 misses. The benefits seen at L1 by these optimizations are only marginal.

All optimizations effectively reduce the total amount of work done and hence they all show an improvement in L1 misses. Label filtering shrinks the R-matrix, by deleting uninformative columns, so that the matrix can, most of the times, fit in few cache lines of L1. DOM reduces the number of data accesses by avoiding unnecessary recursions on the R-matrix. Both LF and DOM jointly improved L1 cache misses by 19 times over the basic algorithm. On top of these, computation chunking completely localizes the process of finding extensions to higher level caches by intelligently grouping the set of computations. On the whole, our fully optimized algorithm *improved the L1 cache performance* of the basic algorithm *by* $1,442$ *times*. Since chunking eliminates the need for large embedding lists, L3 misses and hence the off-chip traffic are also reduced. With multiple processors, we observe similar trends except that the improvement in L2 and L3 miss rates were slightly higher compared to a single node execution. The amount of reduction in cache misses at different levels of the memory hierarchy translates into overall improvement in run time as shown in the figure 4b. Overall, *tree pruning and recoding*, *computation chunking* and also to a degree *simultaneous subtree matching* provide the major benefits in realizing an efficient and memory-conserving frequent subtree mining algorithm.

3.8.2 Analysis of bandwidth pressure

In the next experiment (see Figures 4c & d), we perform a coarse-grained analysis of how off-chip bandwidth varies during the course of execution of the algorithms 3 and 4. We divided the time into small slices with a duration of *one msec*. We used the PAPI native events to instrument the code for measuring the off-chip traffic during each time slice. This is estimated to be the product of L3 line size and the number of L3 cache misses in that time slice.

[4] `http://www.cs.rpi.edu/~zaki/software/` – derived from the domain of web usage mining

[5] `http://icl.cs.utk.edu/papi/index.html`

[6] Performed on a SGI Altix system with 16 1.4 GHz Itanium 2 processors with 32 GB memory. Each processor has L1-data (16KB), L1-instruction (16KB), L2 (256KB), and L3 (3MB) on-chip caches.

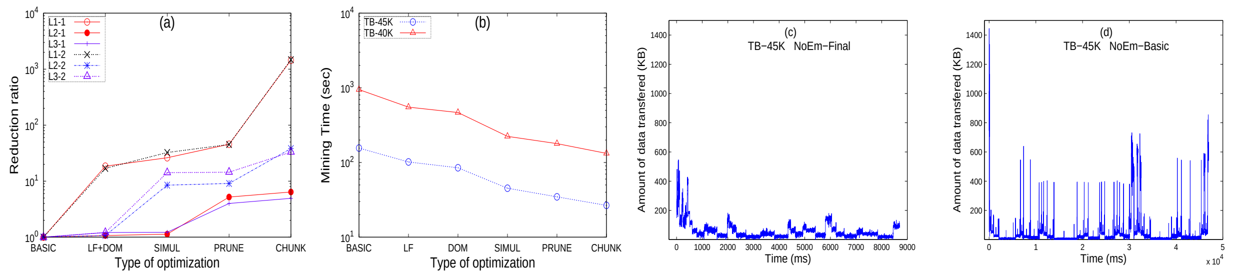


Figure 4: Characterization of optimizations

Though this is rather a coarse-grained analysis, through this experiment, we want to bring out the differences in the main memory accesses made by the strawman and optimized algorithms. Please note that we conducted this experiment with a single processing element and we use this to show that our algorithm reduces the overall off-chip traffic (and its variability) thereby making it suitable multi-core architectures.

Initialization activities like reading the data set result in a large number of L3 misses (cold misses) explaining the initial spikes in these figures. In the left figure, the clusters of lines correspond to the amount of data transfered while mining individual patterns. During the course of mining a single pattern, the main memory accesses made by the fully optimized algorithm is *more uniform and small in size* when compared to the basic algorithm. This is due to the uniform nature of the computation – build a chunk and process it – in algorithm 4. On the other hand, in Algorithm 3, entire embedding list is first constructed showing a spike due to increased off-chip traffic. It then finds the extensions match-by-match showing a sudden dip in the off-chip traffic. After the initialization activities, all the accesses made by the algorithm 4 *are small (below* $200KB$ *per msec)* when compared to *large spikes (around* $500KB$ *per msec)* in the basic algorithm 3. Specifically, while mining the patterns with a huge number of matches, compare the small spikes (similar to the one at 6000 *msec*) in the left figure with the corresponding large spikes (similar to the one around $3*10^4$ *msec*) in the right figure.

Our coarse grained analysis reveals that the optimizations we present seems to be quite effective in *reducing the off-chip traffic*. An added benefit is that the computational chunking also attempts to ensure *uniform and small-sized requests on the front side bus* making the algorithm quite suitable for emerging CMP systems.

3.8.3 Analysis of working set size

In our next experiment we empirically examined the working set size of the fully optimized algorithm. *Cachegrind* [7] is used to perform this study in order to run the algorithm with different cache configurations. We fixed the L2 size and its associativity and we analyzed how L1 miss rate changes as we vary the L1 size from 2 KB to 256 KB. We ran this experiment for different support levels on both the data sets and found that the L1 miss rate falls steeply between 8 KB and 16 KB and stays almost constant after 16 KB. This suggests that the *working set size* is somewhere *between* 8 *KB and* 16 *KB* which is quite small and therefore quite suitable for emerging multi-core systems, which are expected to have very small caches.

4. PARALLEL ALGORITHMS

The algorithms and optimizations described so far speed up the mining process on a single processor. With low support levels and the fact that the subtrees are embedded, even the optimized algorithm can be expensive motivating the need for fast parallel algorithms. Consider the fact that the fully optimized serial algorithm for $TB-30K$ has taken 2,900 seconds to mine the frequent subtrees. In this section, we explore various coarse-grained and fine-grained task and data partitioning techniques to parallelize the algorithm 4. In a broader sense, these techniques are classified as *task-parallel* and *data-parallel* approaches. We consider the work that is done for finding the extensions of a single pattern as a *job* or a *task*. Task-parallel approaches share one or more patterns among multiple processors. In data-parallel approaches, all the processors mine the same pattern in the search space but by looking at a disjoint portion of the data set. The former type of approaches exploits the inter-task parallelism and the latter exploits the intra-task parallelism.

Our parallel algorithms target the systems with multiple processors sharing the main memory. These systems include classic shared memory systems (SMPs) and recent multi-core systems. Our algorithms adaptively modulate the type and granularity of the work being shared among processors. Proposed parallel algorithms differ in the type and the granularity of the work that is being shared. Each processor shares a job by enqueuing it in the global *job pool* from where other "idle" processors can steal the job. If a processor finds the job pool to be empty, it will vote for termination and waits until other processors cast their vote. The waiting processor retracts its vote when it finds one or more shared jobs. The amount of time spent in waiting is denoted as *idle time* I_i. We refer to the standard deviation of I_i's as SD_I and their average as AVG_I. We use SD_I as a measure that provides a sense to the amount of load imbalance. Unless otherwise stated, all the parallel experiments are conducted on a SGI Altix 3000 shared memory system with 32 1.3 GHz Intel Itanium 2 processors and 64 GB of main memory.

4.1 Equivalence class -level partitioning

In algorithm 1, each frequent label f in F1 is a seed pattern that is associated with an equivalence class. The sets of subtrees grown from two different frequent labels are disjoint and hence their equivalence classes can be mined independent of each other. The seed patterns are first enqueued in the job pool and all threads dequeue the jobs and mine them. Note that this approach, as opposed to statically partition-

[7] `http://valgrind.org/info/tools.html`

ing the seed patterns among threads, automatically achieves some level of load balance. For $TB-35K$ on 8 threads, this strategy gave only 7% improvement over the serial algorithm with a SD_I of 233 seconds demonstrating a very high load imbalance.

This strategy does not perform well in the presence of data skew – most of the real data sets are skewed. For example, almost 95% of the time in mining Treebank data set was spent in two equivalence classes and approximately 98% of the mining time was spent in one equivalence class of Cslogs data set.

4.2 Pattern -level partitioning

To achieve a better load balance, one can employ a more fine-grained approach where the work is shared at the level of individual patterns. As soon as a thread generates a new extension point (i.e., a new pattern) it is pushed into the job pool. While this strategy does reduce the idle time to an extent it suffers from memory management and locality issues. Few meta-structures (e.g., *newtidlist*) allocated while mining a parent pattern are shared by all of its child patterns. Since each extension is now put into the job pool, such meta-information has to be replicated and attached with the extension. Alternatively, all child patterns can share the same meta-structure, in which case the last child has to deallocate the meta-structure. Both these methods incur some extra processing overhead and complicates the memory management. Mining child patterns at the processor that created them often results in good locality as the needed data set trees are likely to be in cache. Since this strategy can not guarantee such properties, it may result in poor cache performance.

We expect the idle times with this strategy to be very small but the observed SD_I and AVG_I values, for a $TB-35K$ experiment on 8 nodes, were 24.2 and 36.3 seconds, respectively resulting in a speedup of 4.3. This counter-intuitive result is due to the skew among the individual tasks (see Section 4.5). For these reasons, sharing of child patterns should be avoided as much as possible, as we do in our next strategy.

4.3 Adaptive task partitioning

The first strategy is too conservative in sharing the work and on the other extreme, the pattern-level approach shares every generated pattern even when all the other processors are busy. It is therefore desirable to have a strategy that operates somewhere in the middle-ground and shares the jobs only when there are one or more "idle" processors to steal them. Note that a processor idles only when the job pool is empty. Therefore, we can decide whether to share a job or not by looking at the number of jobs in the job pool. Moreover, instead of waiting till the job pool is empty, we share the jobs as and when the number of jobs in the pool falls below a pre-defined threshold number. Adaptive task partitioning exhibits a good cache performance as the child patterns are usually mined by the processor that created them. For $TB-35K$, this strategy gave a 5.7-fold speedup on 8 processors with SD_I and AVG_I of 17 and 37 seconds, respectively. A non-linear speedup and high idle times are again due to the skew among different tasks (see Section 4.5).

4.4 Data partitioning

First three approaches share the work in terms of one or more patterns. They incur a high load imbalance if the individual patterns are skewed in terms of the mining time i.e., few patterns take up more time compared to others. For example, a particular 6-node pattern in Cslogs data set, has taken approximately 86% of the total serial mining time because of its really large number of matches (see Section 3.7). The resulting load imbalance can be reduced by partitioning the work to be don in mining each individual pattern.

In this approach, all processors, at any time, work on a single pattern by sharing the *newtidlist*. By sharing *newtidlist*, each processor finds the extensions with respect to a subset of trees. A reduction operation is performed at the end to combine the set of all extensions found by each processor. However, a static partitioning of the *newtidlist* can lead to load imbalance as the time spent in processing the tree is not same for each tree – it depends on the number of matches. Therefore, we rely on a more dynamic approach where we treat the set of trees in *newtidlist* as a *tree pool* (analogous to the job pool) and each processor picks up a tree from the pool and finds the extensions with respect to it. All the processors thus share the job of mining a single pattern by dynamically partitioning the associated tree pool. When we applied this strategy for mining the Treebank data set, the parallel efficiency is actually degraded due to increased overhead from the synchronizing reduction operations.

4.5 Hybrid approach

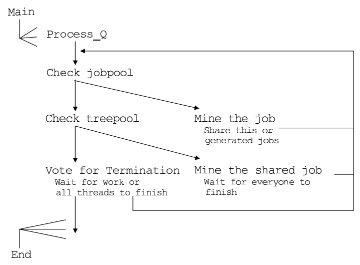


Figure 5: Control flow in the hybrid approach

Our hybrid approach shares *newtidlist* only when there are processors to steal the work. The figure 5 describes the control flow of our parallel algorithm that adopts a hybrid approach. The main thread reads the data set, sets up the job pool, and then spawns the threads. Each thread iteratively look for jobs in the job pool and in the tree pool, if the job pool is empty. If both are empty, it votes for termination and blocks itself until some other thread shares work or until all the other threads vote for termination. Once all threads vote for termination, they join back on to the main thread.

To achieve a better load balance, we enforce a total order on the tree pool before it is set out for sharing. Such a total order hopefully sorts the trees according to the number of matches that they hold. We expect that the number of matches in T is proportional to its size. Therefore, we sort the trees in *newtidlist* based on their size before it is shared among processors. This heuristic is similar to the one employed in classic job scheduling where the jobs are sorted in decreasing order of their processing time.

The job pool can also be maintained at each processor to reduce the contention to the global job pool. On both CS and TB data sets, such a distributed job queue model showed no improvement over the global job queue model.

However, please note that this behavior is data set dependent.

The effect of different load balancing strategies and the resulting speedups are shown in the figures 6 and 7. The hybrid approach realizes a near linear speedup till 12 processors on Cslogs (till 10 on Treebank), after which the speedup saturates due to the trees with huge number of matches. In a $CS-600$ experiment on 8 nodes, we found that a 6-node pattern has approximately 33 million matches in a single 99-node tree whose mining took around 51 *sec*. At 14 processors, the speedup saturates as the lower bound of 51 *sec* is reached. Note that the saturation in case of Treebank is not evident in the graph as the knee of its plateau is at 16 processors. To overcome this type of bottlenecks, one has to design new algorithms, which can operate on a single tree in parallel.

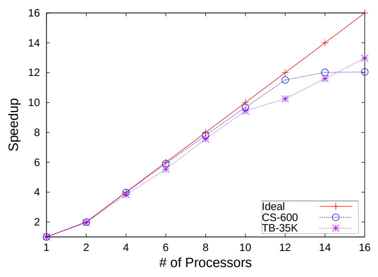


Figure 6: Effectiveness of the hybrid approach

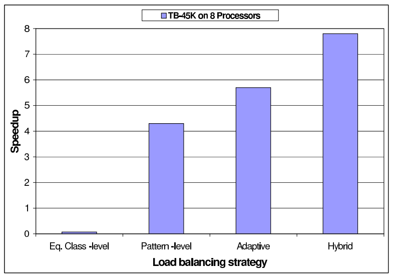


Figure 7: Effect of different load balancing strategies

5. CONCLUSIONS

In this paper, we proposed a new efficient parallel frequent subtree mining algorithm that is suitable for emerging CMP architectures. We improved the state-of-the-art algorithms, TRIPS and TIDES by intelligently avoiding the maintenance of the embedding lists and by leveraging a series of optimizations. We showed that our algorithm outperforms TRIPS by reducing the memory footprint size (up to 366 times) and by improving the run time (up to 4 times). Through a detailed characterization of the proposed optimizations, we showed that our algorithm improves the locality (L1 misses by 1,442 times), keeps small working sets (around 16KB), and makes more uniform and small-sized accesses to the main memory. We proposed various adaptive data and task partitioning strategies for reducing the load imbalance in the presence of data skew. Our evaluation showed that these load balancing strategies achieve near linear speedups up to 13 times on 16 processors.

6. REFERENCES

[1] T. Asai and et al. Efficient substructure discovery from large semi-structured data. *Proceedings the 2nd SIAM International Conference on Data Mining (SDM)*, pages 158–174, 2002.

[2] I.D. Baxter, A. Yahin, L. Moura, M. SantAnna, and L. Bier. Clone Detection Using Abstract Syntax Trees. *Proceedings of the International Conference on Software Maintenance*, page 368, 1998.

[3] G. Buehrer, S. Parthasarathy, and Y. Chen. Adaptive parallel graph mining for cmp architectures. In *Proceedings of the Sixth International Conference on Data Mining*, pages 97–106. IEEE Computer Society, 2006.

[4] Y. Chi and et al. CMTreeMiner: Mining Both Closed and Maximal Frequent Subtrees. *The Eighth Pacific Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*, 2004.

[5] Y. Chi and et al. Frequent Subtree Mining-An Overview. *Fundamenta Informaticae*, 66(1):161–198, 2005.

[6] T.H. Corman, C.E. Leiserson, R.L. Rivest, et al. Introduction to Algorithms. *MIT Press, Cambridge, MA*, 2001.

[7] A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y.K. Chen, and P. Dubey. Cache-conscious frequent pattern mining on a modern processor. *Proceedings of the 31st international conference on Very large data bases*, pages 577–588, 2005.

[8] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. *ACM SIGMOD Record*, 2000.

[9] S. Nijssen and J.N. Kok. Efficient discovery of frequent unordered trees. *First International Workshop on Mining Graphs, Trees and Sequences*, pages 55–64, 2003.

[10] R. Prasad and et al. Annotation and Data Mining of the Penn Discourse TreeBankh. *in Proceedings of the ACL Workshop on Discourse Annotation*, 2004.

[11] R. Praveen and M. Bongki. Prix: Indexing and querying xml using prüfer sequences. In *ACM Transactions on Database Systems (TODS)*. ACM Press, 2006.

[12] U. Ruckert and S. Kramer. Frequent free tree discovery in graph data. *Proceedings of the 2004 ACM symposium on Applied computing*, pages 564–570, 2004.

[13] H. Tan and et al. IMB3-Miner: Mining Induced/Embedded Subtrees by Constraining the Level of Embedding. *The Eighth Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 450–461, 2006.

[14] S. Tatikonda, S. Parthasarathy, and M. Goyder. Lcs-trim: Dynamic programming meets xml indexing and querying. *Technical Report*, 2006.

[15] S. Tatikonda, S. Parthasarathy, and T. Kurc. Trips and tides: New algorithms for tree mining. *Proceedings of the 15th ACM international conference on Information and knowledge management (CIKM)*, pages 455–464, 2006.

[16] A. Termier and et al. Efficient mining of high branching factor attribute trees. *Proceedings of Fifth IEEE International Conference on Data Mining, 2005*, pages 785–788, 2005.

[17] R. Wagner and M. Fischer. The String-to-String Correction Problem. *Journal of the ACM (JACM)*, 1974.

[18] C. Wang and et al. Efficient Pattern-Growth Methods for Frequent Tree Pattern Mining. *The Eighth Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD04)*, 2004.

[19] H. Wang, S. Park, W. Fan, and P. S. Yu. Vist: a dynamic index method for querying xml data by tree structures. ACM Press, 2003.

[20] K. Wang and H. Liu. Discovering typical structures of documents: a road map approach. *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 146–154, 1998.

[21] MJ Zaki. Efficiently mining frequent trees in a forest: algorithms and applications. *IEEE Transactions on Knowledge and Data Engineering*, 17(8):1021–1035, 2005.