Mining Tree-Structured Data on Multicore Systems

Shirish Tatikonda
The Ohio State University
Columbus, OH 43210
tatikond@cse.ohio-state.edu

Srinivasan Parthasarathy
The Ohio State University
Columbus, OH 43210
srini@cse.ohio-state.edu

ABSTRACT

Mining frequent subtrees in a database of rooted and labeled trees is an important problem in many domains, ranging from phylogenetic analysis to biochemistry and from linguistic parsing to XML data analysis. In this work we revisit this problem and develop an architecture conscious solution targeting emerging multicore systems. Specifically we identify a sequence of memory related optimizations that significantly improve the spatial and temporal locality of a state-of-the-art sequential algorithm – alleviating the effects of memory latency. Additionally, these optimizations are shown to reduce the pressure on the front-side bus, an important consideration in the context of large-scale multicore architectures. We then demonstrate that these optimizations while necessary are not sufficient for efficient parallelization on multicores, primarily due to parametric and data-driven factors which make load balancing a significant challenge. To address this challenge, we present a methodology that adaptively and automatically modulates the type and granularity of the work being shared among different cores. The resulting algorithm achieves near perfect parallel efficiency on up to 16 processors on challenging real world applications. The optimizations we present have general purpose utility and a key outcome is the development of a general purpose scheduling service for moldable task scheduling on emerging multicore systems.

1. INTRODUCTION

The field of knowledge discovery is concerned with extracting actionable knowledge from data efficiently. While most of the early work in this field focused on mining simple transactional datasets, recently there is a significant shift towards analyzing data with complex structure such as trees and graphs. This article focuses on mining tree structured data that is useful in a wide range of application domains. For example, the secondary structure of a RNA molecule is often represented as a rooted ordered tree [46]. Uncovering common substructures from a database of such trees helps in discovering new functional relationships among corresponding RNAs [11]. These substructures are known to be useful in predicting RNA folding [15] and in functional studies of RNA processing mechanisms [24]. Similar techniques are also applicable for studying glycan molecules which are responsible for many cellular processes [13], and phylogenies which denote evolutionary relationships among different organisms [25, 42].

In case of web log mining, the visitor accesses to a website can be modeled (with some approximations) as trees [42]. Frequent patterns extracted from these trees can help in making recommendations, in web personalization, and in better organization of web pages [26]. Frequent tree mining is also found to be useful in analyzing XML repositories [44], in designing caching policies for XML indices [41], in designing automatic language parsers [8], in examining parse trees [4], and in many other applications. The essential problem in these instances can be abstracted to the one that of *discovering frequent subtrees from a set of rooted ordered trees* [3, 10, 18, 22, 28, 31, 32, 35, 42] – the focus of this article.

The current explosion in the availability of information necessitates the development of efficient and scalable data mining algorithms which can deal with gigabytes of data. An important strategy here is to leverage recent advancements in computer architecture which are making the computer cycles cheap and abundant. For instance, *multicore* or *chip multiprocessor* (CMP) systems, primarily motivated by power and energy considerations, are becoming extremely common-place. The general trend has been from single-core to many-core: from dual-, quad-, eight-core chips to the ones with tens of cores [1]. For such systems, it is becoming increasingly evident that a memory conscious design is critical to obtain good performance. There is both a need to alleviate the problem of memory access latency as well as to reduce the bandwidth pressure since technology constraints are likely to limit off-chip bandwidth to memory as one scales up the number of cores per chip [14]. Equally important, it becomes imperative to identify scalable and efficient parallel algorithms to deliver performance commensurate with the number of cores on chip. A fundamental challenge is to ensure good load balance in the presence of data and workload skew pointing to the need for an adaptive design strategy.

We contend, and later demonstrate through a detailed performance study, that extant tree mining algorithms require significant changes to meet these challenges. The rationale is as follows. *First*, they all trade space for improved execution time by employing several potentially large data structures. Such strategies developed for unicore systems with large memory are likely to be inefficient on multicores where the premium on off-chip memory accesses is expected to be very high. Additionally, parallel instantiations of such algorithms will require shared access to large data structures and often dictate housing additional redundant information thereby reducing the overall efficiency. *Second*, even if the first issue can be resolved through appropriate memory conscious designs, one still needs to develop an effective parallelization strategy account-

[1] `http://techfreep.com/intel-80-cores-by-2011.htm`

VLDB '09, August 24-28, 2009, Lyon, France

ing for workload skew. Moreover, there is also a need to expose and subsequently exploit a fine-grained parallelism on such architectures [23]. In this article we address these challenges and make the following contributions.

- We propose several **generic memory conscious optimizations** which alleviate the problem of memory access latency as well as reduce the bandwidth pressure on the front side bus. Specifically, optimizations that limit the pointer use, leverage a novel compressed representation of the problem space, and enable computational chunking have been designed.

- Through a detailed characterization, we demonstrate that our optimizations **reduce the memory usage by up to** 366**-folds** while improving the run time by four times when compared to state-of-the-art. Through a novel bandwidth measurement strategy, we also show that they make uniform and small sized memory requests, resulting in a **reduced bandwidth pressure** on the front side bus.

- We empirically show that these optimizations are necessary but not sufficient for efficient parallelization on multicores. We then present a multi-level parallel algorithm that **automatically and adaptively** modulates the type and granularity of the work, as dictated at run time by the input parameters and data set properties. Our algorithm leverages a **general purpose scheduling service** we have developed for emerging multicore systems.

- We show, on a dual quad core CMP system and on a 16-processor SMP system, that our load balancing strategies achieve **near perfect parallel efficiency** on challenging real world data sets.

The rest of the article is organized as follows. We first define the problem and show the limitations of existing works in Section 2. We then present our memory optimizations in Section 3. Section 4 describes both our parallelization strategies and our scheduling service designed for multicores. Results from empirical evaluation are shown in Section 5 and Section 6 demonstrates the broader applicability of our contributions in the paper.

2. BACKGROUND AND CHALLENGES

DEFINITION 2.1. ***Frequent Subtree Mining****: Given a database of rooted ordered trees, enumerate the set of all frequent embedded subtrees (|FS|) i.e., the subtrees whose support is greater than a user defined minimum support threshold.*

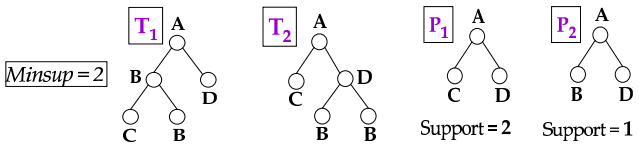

Figure 1: Example database and patterns

The minimum support ($minsup$) can be expressed either as a percentage or as an absolute number of database trees. There are two ways to define the *support* of a subtree (or pattern) S – *transaction-based* and *occurrence-based*. The former counts the number of trees in which S occurs, and the latter counts the total number of *embeddings* (or matches) in the database. If S occurs twice in a given tree then its transaction support is 1 whereas its occurrence support is 2. In this article, we use the transaction-based definition.

	TreeMiner	iMB3-T	Trips
Working set[1] (KB)	256	128	64
Memory usage[2] (GB)	7	32	4

[1]On Treebank data set at $minsup$=$45K$ (85%) – see Section 5

[2]Maximum memory footprint observed in *all* our experiments in Section 5

Table 1: Characterization of Tree Mining Algorithms

In Fig. 1, P_1 has one embedding (matching) in each of T_1 and T_2, whereas P_2 occurs only in T_1 with 2 embeddings. If $minsup$=2 then only P_1 considered *frequent*. A variant of this problem mines for *induced*, as opposed to *embedded*, subtrees [2].

Any mining process has two phases, *candidate generation* and *support counting*. The first one generates candidate subtrees which are evaluated for their frequency in the second one. The challenges in two phases are to efficiently traverse the search space and to perform subtree isomorphisms, respectively. We employ a pattern-growth approach where a frequent subtree S is repeatedly *grown* with new edges to yield new candidate subtrees. The new edge is called an *extension*, and the extension process is called *point growth*. An equivalence class of S (denoted as $[S]$) contains all subtrees generated from S through *one or more* point growths. If S is a single node v then $[S]$ has all the subtrees whose root is v.

Related Work: A majority of extant tree mining algorithms employ special data structures called as *embedding lists* (EL) to store extra state with which they avoid repeated executions of expensive subtree isomorphism checks. All matches of a frequent subtree S are stored in its EL so that the matches for subtrees grown from S can be found easily. *TreeMiner*, proposed by Zaki, stores the matches in *scope-lists* whose entries (in a worst case) are of size equal to the pattern size [42]. They usually *occupy a lot of memory* due to redundant information (see Table 1), especially when the number of overlapping matches is high – a common case in most real-world data sets. New subtrees are generated by *joining* these large lists, resulting in expensive run time performance.

iMB3 proposed by Tan *et al.* uses *occurrence lists* to store the embeddings [28]. It also maintains a *dictionary* for representing the data and *descendant lists* to track *all* descendants of a frequent node, which are *persistent* across entire execution. The memory usage is thus very high even at moderate support values (see Table 1). They recently developed a similar method that uses transaction-based support, which hereinafter, is called as *iMB3-T*.

Wang *et al.* proposed *Chopper* and *XSpanner* [35]. Chopper recasts the subtree mining into sequence mining but suffers from large number of false positive subsequences. XSpanner, in contrast, employs recursive projections which are often too complex and result in pointer-chasing [31], amounting to poor performance.

Researchers have also proposed methods like CMTreeMiner [10] and PathJoin [39] which reduce the output size by mining closed and maximal subtrees. They however address only induced subtrees. Another method by Termier *et al.* [33] assume that no two sibling nodes can have the same label – an unrealistic assumption. There exist several other algorithms which differ in the type of subtrees that they mine [3, 18, 22, 25]. Please refer to the survey by Chi *et al.* for more details [9].

In this work we present our memory optimizations in the context of Trips [31]. Of all the algorithms discussed thus far it has the smallest memory footprint and working set (see Table 1), and is most efficient (see Section 5). While these numbers appear quite reasonable for Trips, at lower supports, they can still be much too large. We next briefly describe Trips as Algorithm 1.

[2]An induced subtree preserves parent-child relationships whereas an embedded subtree preserves ancestor-descendant relationships.

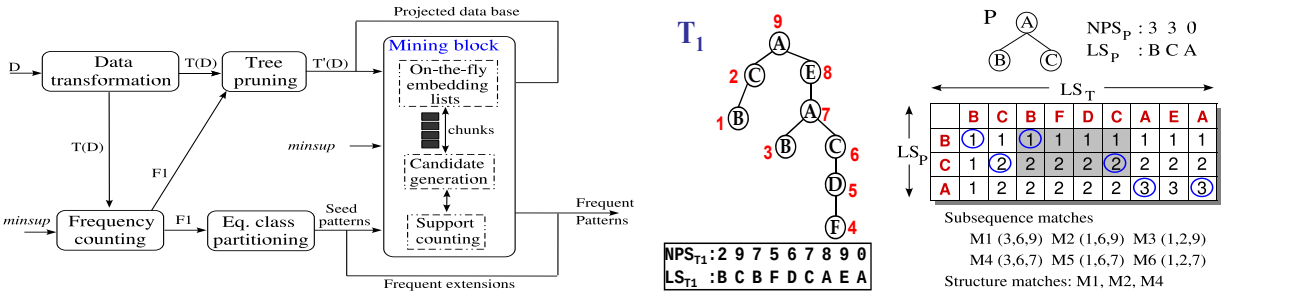

Figure 2: (a) Framework of our MCT algorithm (b) Example database tree, pattern, R-Matrix

Trips [31]: It encodes each database tree T as two sequences: Numbered Prüfer Sequence NPS_T; and Label Sequence LS_T. They are constructed iteratively based on post-order traversal numbers (PON). In every iteration, the node (say, v) with the smallest PON is removed. The label of v is appended to LS_T, and the PON of v's parent is added to NPS_T (see Fig. 2b). Note that this representation is different from a similar ordering used by Rao *et al.* [21]. The differences are detailed elsewhere [30].

Algorithm 1 Trips Algorithm

Input: $\{T_1, T_2, \ldots, T_N\}, minsup$
(D, F_1) = Transform(T_i): $1 \leq i \leq N$
for each f in F_1 **do**
  mineTrees ($NULL$, (f, -1), D)

mineTrees (pat, extension (lab,pos), $tidlist$)

1: $newpat \leftarrow pat + (lab, pos)$ // pattern extension
2: output $newpat$
3: **for each** T in $tidlist$ **do**
4: **if** (lab, pos) is an extension point for pat in T **then**
5: update the embedding list of T i.e., $EL(T)$
6: add T to $newtidlist$ // indicates $newpat$ occurs in T
7: H = NULL
8: **for each** T in $newtidlist$ **do**
9: **for each** node v in T **do**
10: **for each** match m in $EL(T)$ **do**
11: **if** v is a valid extension to m **then**
12: add the extension point to H
13: **for each** ext in H **do**
14: **if** ext is frequent **then**
15: mineTrees ($newpat$, ext, $newtidlist$)

Given database is first transformed into sequences (D) and all frequent nodes (F_1) are found. For each $f \in F_1$, *mineTrees* is called to mine subtrees from its equivalence class $[f]$. *mineTrees* is a recursive procedure with three inputs: a pattern pat, an extension (lab, pos), and a projected database $tidlist$. An extension (lab, pos) of pat defines a new subtree $newpat$ (in line 1) obtained by attaching a node with label lab to a node in pat whose PON is equal to pos – each extension uniquely identifies a subtree grown from a given pattern. The *projected database* (PD) $tidlist$ contains the list of trees in which pat has *at least one* embedding.

Initially, EL(T) contains all matches of pat in T. Lines 3-5 transform EL(T) into a list for $newpat$ by appending the positions in T at which (lab, pos) match – equivalent of finding isomorphisms of $newpat$ in $tidlist$. Line 6 builds the new projected database $newtidlist$. When *mineTrees* returns, the newly appended entries are deleted so that only the matches of pat remain in EL(T).

For every $T \in newtidlist$, we evaluate each node in T against all embeddings of $newpat$ in EL(T) to discover new extensions (lines 8-10). Resulting extensions are hashed into H which records their frequencies (lines 11-12). Lines 13-15 process the *frequent* extensions for producing larger subtrees. Note that each extension in H denotes a unique subtree that is grown from $newpat$.

Parallelization of semi-structured data mining algorithms is severely limited by the existence of embedding lists (see Section 5.3). Therefore, simple parallel algorithms developed for itemsets [43] (which do not involve such lists) are not generalizable for trees and graphs. Buehrer *et al.* have parallelized graph mining workloads on CMPs but their focus was primarily on harnessing the tradeoff between time and space [6], which is critical for graph mining since the subgraph isomorphism is very expensive. In contrast, our optimizations improve both the run time and memory performance of Trips (see Section 5). Furthermore, our parallel strategies operate at much finer level of granularity than the ones used by Buehrer *et al.* Other parallel algorithms developed for data mining tasks like clustering [20] and classification [45] focus on shared memory SMP systems and shared-nothing cluster systems, and they are not readily applicable to CMPs. The research on exploiting multicores for data analysis is still in its early stages and much needs to be done. We believe that this is the first attempt to parallelize tree mining workloads on any type of architecture. Further, we develop a fine-grained strategy that finds matches from a single tree, in parallel – a first of its kind, to the best of our knowledge.

Recently, several software frameworks like Google's Map/Reduce, open source Hadoop, and Microsoft's Dryad have been proposed for scalable analysis of large amounts of data on clusters of computers. They primarily address *data-parallel* applications and may not be suitable for a variety of highly irregular data mining applications where a combination of *task-parallel* and *data-parallel* approaches is essential. In the advent of multicores, there is some effort in designing new libraries, programming languages, compilers, and tools like Intel's TBB [3] and Microsoft's PCP [4]. They do not address the same problem as we do. For example, Intel's TBB and related libraries in PCP (designed for Visual Studio) provide new programming interfaces similar to pthreads and OpenMP for easier development of scalable and portable applications. They do not quite support the type of adaptive scheduling service that we describe in Section 4.5.

Before we present our algorithms, we briefly discuss the challenges in dealing with CMP systems. *First*, applications must *control the memory usage* as large footprints not only force OS to rely on virtual memory but also increase the bus contention – likely to be severe on CMPs since all cores share a common memory bus. *Second*, algorithms must exhibit *good cache locality* and maintain *small working sets* because in future multicore systems the contention for on-chip caches is likely to increase [14]. Achieving

[3] `http://www.threadingbuildingblocks.org/`
[4] `http://msdn.microsoft.com/en-us/concurrency/`

good spatial and temporal locality in pattern mining algorithms is very difficult because of pointer-based data structures and huge search space, respectively. *Third*, one must efficiently address the issue of *load balance* for good scalability. Highly irregular nature of pattern mining workloads makes the task estimation very difficult. There is also a need for algorithms which expose and subsequently exploit fine-grain parallelism for multicore systems [23]. We now present our memory optimizations and load balancing techniques to address these challenges.

3. MEMORY OPTIMIZATIONS

Though embedding lists (EL) are designed to trade space for improved execution time, they can grow arbitrarily in size, especially at low support values. Consider the embedding lists in Trips (see Algorithm 1). Assume a worst case scenario of a *chain tree* (a path) of size n, where every node has the same label (say, A). For a single node pattern, EL would contain exactly $\binom{n}{1}$=n entries. When it is extended to an edge A-A, the list will contain $\binom{n}{1}$ + $\binom{n}{2}$ = $\frac{n(n+1)}{2}$ entries. Similarly, when the pattern has n nodes (i.e., the complete path), the number of entries in EL is equal to $\sum_{i=1}^{n}\binom{n}{i}$ = $2^n - 1$, even though there is exactly a single embedding for the pattern. The size of EL thus *increases proportionally with the number of matches*, which is exponential in a worst case. Such cases often occur in real-world data sets (see Section 5).

The architecture of our Memory Conscious Trips (MCT) is shown in Figure 2. Tree database D is first transformed into sequences (T(D)) – see Section 2. Infrequent nodes are then pruned from T(D) to produce T′(D) (see [29] for details). Both T′(D) and the set of frequent nodes $F1$ are fed to the mining block with three phases: on-the-fly embedding lists OEL (see Section 3.1), candidate generation CG, and support counting SC. Instead of storing the embedding list, CG invokes OEL to compute the matches on-demand (see Section 3.3). Produced matches are processed by CG and SC to produce frequent extensions. Generated extensions are fed back to the mining block to yield larger patterns.

3.1 On-the-fly Embedding Lists (NOEM)

In MCT, instead of storing embedding lists (EL) explicitly, we adopt a strategy that dynamically *constructs* the list, *uses* it, and then *de-allocates* it. In graph-theoretic terms, constructing a dynamic EL is equivalent of finding the set of all (embedded) subtree isomorphisms of a given pattern in the database – a core problem in XML indexing. We construct EL on demand by employing a dynamic programming based approach that is inspired by recent research in XML indexing [30, 47]. There are however some important differences – (i) in XML indexing, there is no notion of embedding lists which are employed to save time on repeated subtree isomorphisms, (ii) each mining run here comprises of *many* tree matching queries, and our subsequent optimizations. Unlike in XML indexing, a straight application of these techniques in fact increases the run time (see Section 5). We devise techniques to improve the performance by reorganizing the computation (see Section 3.3). Note that dynamic list construction affects only the lines 3-6 of Alg. 1 – correctness of the algorithm is *still intact*.

Say, we need to find matchings of subtree S=(LS_S, NPS_S) in a tree T=(LS_T, NPS_T). Let $|S| = m$ and $|T| = n$. Prüfer sequences, due to the way they are constructed, possess an important property that if S is an embedded subtree of T then the label sequence LS_S is a subsequence of LS_T. i.e., being a subsequence is a *necessary but not sufficient* condition for subtree isomorphism.

First, we check if LS_S is a subsequence of LS_T or not by computing the length of their longest common subsequence (LCS) using a traditional dynamic programming approach [34] (see Alg. 2). It constructs a matrix R using Equation 1 so that the length of LCS is given by the matrix entry $R[m, n]$. If $R[m, n] \neq m$ then we conclude that S is *not* a subtree of T (see Fig. 2b).

Algorithm 2 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**
else processR (m, n, 0)

processR (p_i, t_j, L)
1: **if** p_i=0 or t_j=0 **then return**
2: **if** $L = m$ **then**
3: **if** SM[..] corresponds to a subtree **then**
4: update $EMList[T]$ with SM
5: **return**
6: **if** $LS_P[p_i] = LS_T[t_j]$ **then**
7: $SM[m - L] \leftarrow t_j$
8: processR ($p_i - 1$, $t_j - 1$, $L + 1$)
9: processR (p_i, $t_j - 1$, L)
10: **else if** $R[pi, t_j - 1] < R[p_i - 1, t_j]$ **then**
11: processR (p_i, $t_j - 1$, L)

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

Second, if LS_S is a subsequence of LS_T then we enumerate all subsequence matches of LS_S in LS_T by backtracking from $R[m, n]$ to $R[1, 1]$ (lines 6-11 in Alg. 2). A subsequence match SM is denoted by (i_1,...,i_m), where i_k's are the locations in T at which the match occurs i.e., $LS_P[k]$=$LS_T[i_k]$ for $1 \leq k \leq m$ (see Fig. 2b). It is worth noting that, unlike in classical sequence matching problem, here we are interested in obtaining all matches. Since backtracking is performed in backwards, the matches are generated from *right-to-left*.

Third, we filter the false positive subsequences by matching the structure (given by NPS) of SM=(i_1,...,i_m) with that of S (Line 3 in Alg. 2). Such a structural match (map) maps every parent-child relation in S into an ancestor-descendant relation in SM i.e., in T. We first set $map[m]$=i_m (root node). For k = m-1...1, we check if $map[NPS_S[k]]$ is either equal to $NPS_T[i_k]$ or is a nearest mapped ancestor of $NPS_T[i_k]$ – i.e., parent of k^{th} node in S is mapped to an ancestor of i_k^{th} node in T. Since nodes are considered in reverse post order, structure match is also established from right-to-left (i.e., root-to-leaf). Resulting match is finally added to the dynamically constructed embedding list (Line 4 in Alg. 2).

Example: In Fig. 2b, only $M1$, $M2$, and $M4$ are subtree matches. For $M3$: at k=3, the root node is mapped to node i_3=9 in T i.e., map[3]=9. At k=2 ($NPS_S[k]$=3), we set $map[2]$=i_k=2 because map[3]=$NPS_T[i_k]$. However at k=1 (i_k=1), $map[3] \neq NPS_T[i_k]$ and map[3] is not the nearest mapped ancestor of i_1 in T. Since the check fails, $M3$ is declared as a false positive. For $M5$ and $M6$, the check fails at k=1 and k=2, respectively.

3.2 Tree Matching Optimizations

The following three optimizations reduce the amount of redundant computations in Alg. 2. The first two reduce the recursion overhead incurred while backtracking whereas the third one reduces the overhead due to false positives.

1) Label Filtering (LF): Before constructing the R-matrix, we remove those nodes in T which do not appear in S. In Figure 2b, the columns corresponding to nodes D, E, and F can be safely deleted as they do not help in establishing the subsequence match.

2) Dominant Match Processing (DOM): Observe that a subsequence match is established only at the entries (called as *dominant matches*) where both LS_S and LS_T match (condition 2 in Eq. 1). Backtracking on rest of the entries is redundant and must be avoided. In Fig. 2b, dominant matches are encircled. For example, $R[2,6]$ and $R[1,3]$ are dominant and all the other shaded cells simply carry LCS value from one to the other. Recursion from $R[2,6]$ can directly jump to $R[1,3]$ avoiding all the other shaded cells.

3) Simultaneous Matching (SIMUL): Here, we leverage the fact the both subsequence and structure matching phases operate from right-to-left in reverse post order. Therefore, instead of performing the structure matching *after* generating *all* subsequence matches, we can do both the matchings simultaneously. As soon as a subsequence match is established at position k, we perform the structure match at that position. Such an *embedding of structural constraints* into subsequence matching detects the false positives as early as possible and *never* generates them completely.

3.3 Computation Chunking (CHUNK)

Since the size of EL is proportional to the number of matches, the dynamic embedding lists can grow exponentially, in the worst case. This optimization completely eliminates the lists by *coalescing both tree matching and tree mining* algorithms. It operates in three steps: *loop inversion*, *quick checking*, and *chunking*. The computation in Algorithm 1 is reorganized by inverting the loops in lines 9-10 i.e., T is scanned for each match m instead of processing m for each node in T. The second step *Quick checking* notes that the extensions associated with two different matches m_i and m_j ($i < j$) are *independent* of each other. Thus, m_i need not wait till m_j is generated and thus it *need not be stored* explicitly in EL. Finally, *chunking* improves the locality by grouping a fixed number of matches into *chunks*. The tree T is then scanned for each chunk instead of for each match m. Once the extensions against all the matches in one chunk are found, we proceed to the next chunk. This optimization implicitly leverages all the other optimizations described above. Even though it appears to be similar to tiling [38], there are several fundamental differences [29]. In our empirical study, we define chunks to contain 10 matches.

The complete Memory Conscious Trips (MCT) is shown as Algorithm 3. Since it always keeps a fixed number of matches in memory, MCT maintains a *constant-sized* memory footprint throughout the execution. Further, chunking *localizes* the computation to higher level caches, improving both locality and working sets.

Complexity analysis: Like other pattern mining algorithms, MCT belongs to *#P complexity class* as it has to count and enumerate all frequent subtrees. *mineTrees* in Algorithm 3 is invoked exactly once for every frequent pattern (pat+e) that is discovered. For a given S and T (of sizes m and n), the maximum number of recursions on *processR* ($c_{m,n}$) can be approximated as follows [30]:

$$c_{m,n} = \begin{cases} 1 + \sum_{i=1}^{n-m+1}(1 + c_{m-1,n-1}), & if\ n > m \\ n, & if\ n = m \vee m = 1 \end{cases} \quad (2)$$

$c_{m,n}$ has a closed form of $\binom{n+1}{n-m+1}$. The branch conditions in lines 12 and 14 take constant time and the run time of lines 2-10 is governed by the number of matches for S in T.

Algorithm 3 Memory Conscious Trips (MCT)

mineTrees (pat, extension e, tidlist)
A: **for each** T in tidlist **do**
B: construct R-Matrix for T and $newpat$
C: processR (m, n, m)
D: **for each** ext in H **do**
E: mineTrees ($newpat$, ext) recursively

processR (p_i, t_j, L)
1: **if** p_i = 0 or t_j = 0 **then return**
2: **if** L = 0 **then**
3: add SM to $EMList$ and add T to $newtidlist$
4: **if** $|EMList|$ % 10 = 0 **then**
5: **for each** match m in $EMList$ **do**
6: **for each** node v in T **do**
7: **if** v is a valid extension with m **then**
8: add the resulting extension to H
9: $EMList \leftarrow null$
10: **return**
11: **for** $k = t_j$ to 1 **do**
12: **if** $R[p_i][k]$ is dominant & $R[p_i][k]$=L **then**
13: $SM[k] \leftarrow (LS_T[t_j], NPS_T[t_j])$
14: **if** agreeOnStructure (P, SM, k) **then**
15: processR ($p_i - 1$, $t_j - 1$, $L - 1$)

4. ADAPTIVE PARALLELIZATION

We now consider the parallelization of MCT for multicore systems. Note that directly parallelizing Trips algorithm is the first approach we considered. However, embedding lists led to a large memory footprint resulting in significant contention overhead and pressure on the front-side bus. The inherent dependency structure of lists pose difficulties in sharing them, leading to a coarse grained work partitioning and poor load balance (see Section 5.3). Essentially, *parallelization without identifying the memory optimizations*, presented in the previous section, *is extremely inefficient*.

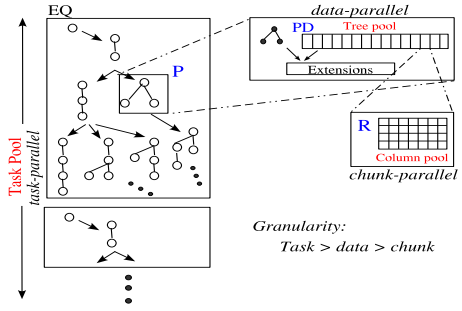

Figure 3: Schematic of different job granularities

Our parallel framework employs a *multi-level work sharing* approach that *adaptively modulates the type and granularity* of the work that is being shared among threads. Each core C_i in the CMP system runs a single instantiation (i.e., a thread) of our parallel algorithm. Henceforth, the terms *core*, *thread*, and *process* are used interchangeably, and are referred by C_i. A *job* refers to a piece of work that is executed by any thread. The set of all threads consume jobs from a *job pool* (JP) and possibly produce new jobs into it. The jobs from a job pool are dequeued and executed by threads on a "first come first serve" basis.

Control flow: As pointed out by Leung *et al.* [16], if the threads are allowed to share the work asynchronously then detecting a global termination would be non-trivial – since the jobs could be shared while a termination detection algorithm is being executed. Instead,

we implement a simple *lock-based* algorithm that is driven by the amount of remaining work in the system. Whenever a thread C_i finds the job pool to be empty, it votes for termination by joining the *thread pool* (TP), and detaches itself (i.e., blocks itself) from execution. Each thread monitors TP at *pre-set points* during its run time, and if it is not empty then it may choose to fork off new jobs onto JP, and notify the threads waiting in TP. The mining process *terminates* when *all* threads vote for termination. We implemented TP using simple locks (akin to semaphores) and condition variables. Similar strategy can be used when multiple job pools are maintained based on thread groups (e.g., distributed and hierarchical job pools) – job pools here act as implicit channels for communication between running and waiting threads.

In our multi-level approach, threads operate in three different levels. Each level corresponds to a different execution mode, which dictates the type and granularity of the jobs in that mode. The three execution modes are *task-parallel*, *data-parallel*, and *chunk-parallel*. The first one exploits the parallelism across different portions of the search space. The data-parallel mode parallelizes the work required to mine a single pattern. Finally at the finest level of granularity, the chunk-parallel mode obtains the matches of a pattern within a single tree in parallel. For a simpler design, we used different job pools for different modes: *task pool* (JP_T), *tree pool* (JP_D), and *column pool* (JP_C), respectively [5]. Shared access to these pools is protected using simple locks. Jobs in these job pools are uniquely identified by *job descriptors*. Each job descriptor J is a 6-tuple as shown below.

$$J = (J.t, J.i, J.f, J.c, J.o, J.r)$$
$$J.t = \left\{ \begin{array}{ll} task, & if \;\; J \in JP_T \\ data, & if \;\; J \in JP_D \\ chunk, & if \;\; J \in JP_C \end{array} \right\}$$

Job type $J.t$ corresponds to the execution mode, and it defines

Algorithm 4 Parallel Tree Mining

```
 1: initialize( ) // I1
 2: identifyGranularities( ) // I3, I4, I5
 3: while true do
 4:   if JP_T is empty then
 5:     if JP_D is empty then
 6:       if JP_C is empty then
 7:         vote for termination
 8:         block itself from execution
 9:         if { all threads voted } break
10:       else
11:         process JP_C // chunk-parallel (I8, sync)
12:     else
13:       process JP_D // data-parallel (I8, sync, I9)
14:   else
15:     mine a task from JP_T // task-parallel (I8, I9)
16: finishUp( ) // I2
```

the remaining entries. Given J, a thread starts with the inputs $J.i$, applies the function $J.f$ to produce an output $J.o$. The control is then returned to the job that created J if return flag $J.r$ is set to *true*. A condition $J.c$ is evaluated at pre-set points to determine whether or not to spawn new jobs from J.

$J.t$ also determines the type of new jobs which J can spawn. A task-level job can either create new tasks or a single job of type $data$. A chunk-level job in JP_C can only be created by a data-parallel job in JP_D. And, jobs in JP_C can not create new jobs i.e., $\forall J \in JP_C$, $J.c = false$. The granularity of jobs in JP_T is more than that in JP_D, which in turn is greater than the granularity of jobs in JP_C. We integrate different execution modes and termination detection as shown in Alg. 4. Such a design *adaptively adjusts the granularity* by switching between the execution modes.

[5] Alternatively, one can implement it as a single job pool with prioritized jobs.

4.1 Task-parallel mode

In this mode, each thread processes jobs from the task pool JP_T where each *task* corresponds to the process of mining full or a portion of an equivalence class $[S]$. Therefore, every job $J \in JP_T$ is associated with a subtree $J.i$=S. The output $J.o$ is the set of subtrees produced from S by invoking $J.f$ ($mineTrees$ in Alg. 3). Further, $J.r$ is always set to *false* in this mode.

Each strategy in this mode differs in the way the search space is *partitioned* into tasks. A naive strategy is to partition the space by equivalence classes – EQ in Figure 3, and schedule different classes (F1 in Alg. 1) on different cores. More precisely,

$$JP_T = \{J \mid J.i \; is \; a \; seed \; pattern \wedge J.c = false\}$$

Since $J.c$ is set to *false*, each job is processed till its completion to produce all subtrees from the equivalence class of seed pattern $J.i$. Such a coarse grained strategy, which is referred to as **Equivalence class task partitioning (EqP)** [43], likely to perform poorly because most real-world data sets are highly skewed and the variance in $|J.o|$'s is usually high.

Another strategy is to partition the search space such that each pattern is treated as a different job – P in Figure 3. Each extension that is produced is enqueued into the job pool as new tasks (i.e., $J.c$ is a tautology). Such a technique is referred to as **Pattern-level task partitioning (PaP)** [43]. It can be formally denoted as:

$$JP_T = \{J \mid J.i \; is \; a \; frequent \; subtree \wedge J.c = true\}$$

Here, JP_T is initialized with frequent nodes from $F1$. If $|\text{F1}| < |C|$ then it is initialized with frequent edges. One can continue to mine in levels until $|JP_T|$ is sufficiently greater than $|C|$. For better efficiency, the projected database of the subtree is also included in $J.i$. This strategy suffers from locality issues since the subtrees may not be mined at the place they were created. Also, aggressive job sharing often results in memory management and computation overheads, motivating the need for an adaptive approach.

In an **adaptive task partitioning (AdP)** strategy, the search space is partitioned *on demand*. New tasks are created only when there are idle threads waiting (for work) in the thread pool TP. Unlike EqP and PaP, this method *adaptively modulates* the task granularity at run time. It can be described as:

$$JP_T = \left\{ \begin{array}{l} J \mid J.i = a \; frequent \; subtree \wedge \\ \quad J.c = (TP \neq \Phi \wedge |Ext| \geq 1) \end{array} \right\}$$

|Ext| is the number of extensions that are *yet to be* processed. Note that, $TP \neq \Phi$ implies that the job pool is empty i.e., new jobs are created only if the job pool is empty and some threads are in wait state. Instead, one can choose to spawn new jobs when the size of the job pool falls below a pre-defined threshold value. The spawning condition in this strategy is evaluated before processing each extension, between lines D-E of Alg. 3. Since it dynamically modulates the task granularity, it not only *achieves good load balance* but also exhibits *good locality* since extensions are mined, whenever possible, on the processor that created them.

4.2 Data-parallel mode

The task partitioning strategies primarily process the search space, in parallel. They do not take the underlying *data distribution* into account. For example in case of a website, one access pattern P_1

can be more dominant and popular than another pattern P_2. Task-parallel strategies *can not* exploit this difference as they implicitly assume that all patterns are of similar complexity. Efficiency can be improved by *dividing* the work associated with the popular i.e., more expensive pattern P_1.

We parallelize the job of mining a single subtree S by looking at its projected database PD_S, 8-12 in Alg. 1 (PD in Figure 3). We treat each tree in PD_S as a different job, and schedule them on to different cores. The pool of database trees JP_D can be denoted as:

$$JP_D = \{J \mid J.i = T : T \in PD_S \wedge J.c = false \wedge J.r = true\}$$

Note that *all jobs* in JP_D (unlike JP_T) are defined in the context of a subtree (S) that is currently being mined. The trees in PD_S are processed simultaneously by multiple cores. Each core produces a subset of extensions, which are then combined to produce a final set of extensions for S – $J.r$ is set to *true*.

We devise an adaptive strategy by combining this basic method that takes the data distribution into consideration with the best task partitioning strategy AdP. It is called as **Hybrid work Partitioning (HyP)**. Here, a core C_i that is currently mining a task-level job $J \in JP_T$ with $J.i$=S forks off new jobs on to JP_D *only when* it finds any idle threads *while* finding extensions from S. Once all trees in JP_D are processed, the core C_i performs a reduction operation to combine the partial sets of extensions. If needed, J *may* now proceed to create new tasks according to *AdP*. Therefore, a task-level job may either create new tasks or new jobs of type $data$ – spawning condition thus needs to be augmented as follows.

$$\forall J \in JP_T,$$
$$J.c = \left\{ \begin{array}{ll} add\ tasks\ to\ JP_T, & \text{if } TP \neq \Phi \wedge |Ext| \geq 1 \\ add\ jobs\ to\ JP_D, & \text{if } TP \neq \Phi \wedge \frac{c(J.i)}{s(J.i)} < \theta \end{array} \right\}$$

While the first condition is evaluated between lines D-E of Alg. 3 (same as AdP), the second one is checked between lines A-B. The second condition governs the creation of data-parallel jobs and it depends on the amount of work that is remaining to complete the task $J.i$. A rough estimate for the amount of remaining work is given by $\frac{c(J.i)}{s(J.i)}$, where $c(J.i)$ is the number of matches found so far and $s(J.i)$ is the support of $J.i$ (known from line 14 in Alg. 1). If this ratio is smaller than a threshold θ (we use $\theta = 20\%$ in our evaluation) then it means that there is a lot of work to be done, and can be shared with others. Such a method essentially decides whether it is worth dividing the work into jobs of finer granularity.

Once the tree pool is created, we sort the trees in the decreasing order of their size. This is similar to classical job scheduling where the jobs are sorted in the decreasing order of their processing time. We sort based on tree size because the mining time that depends on the number of matches in a given tree is *likely* to be proportional to the tree size.

4.3 Chunk-parallel mode

Even the hybrid strategy HyP may not always achieve full efficiency in practice. This is because the trees themselves can be skewed. For example in Bioinformatics, one Glycan or RNA structure may be very large when compared to the other. Such large trees and the trees with large number of matches will introduce load imbalance while using HyP. To deal with such a skew, the job of mining a single tree i.e., the process of finding matches and corresponding extensions from a given tree should be parallelized. This fine grain parallelism is obtained by parallelizing at the level of *chunks*, which are generated in lines 3-4 of Alg. 3. Since chunks are created from individual columns of the R-matrix, we treat each column as a separate job and schedule them on to different cores.

This mode is entered only when all the available parallelism in data-parallel mode is fully exploited. A job of type $data$ in JP_D switches to this mode based on the following condition:

$$\forall J \in JP_D,$$
$$J.c = \{\ spawn\ jobs\ onto\ JP_C, \quad if\ TP \neq \Phi\ \}$$

One can also design $J.c$ based on pattern size, number of matches found so far, and the portion of R-matrix that is yet to be explored. This condition is evaluated between lines 13-14 of Alg. 3.

For each job J in column pool JP_C, the input is a column from R-matrix, and the partial match that is constructed so far (by J's parent job in JP_D). $J.f$ backtracks from the input column to discover the remaining part of the match, and extensions from that match ($J.o$). $J.r$ in this mode is always set to *true* so that extensions generated from different column jobs can be combined at the parent job. Also, $J.c$ is always set to *false*.

4.4 Cost analysis

A key factor to the performance of our parallel framework is the amount of overhead incurred in creating, sharing, and managing jobs and job pools. This overhead is minimal due to following reasons: (i) we avoid any type of meta data structures, making it is easy to fork off new jobs from current computation; (ii) all jobs have very small sized inputs (a small pattern, a tree id, or a column id), and so it is easy to create and share them; (iii) all jobs are shared using simple queueing and locking mechanisms; and (iv) all job spawning conditions can be evaluated in constant time.

Another source of overhead is the number of context switches between different execution modes. We now develop some theoretical bounds on that number by analyzing various job spawning conditions. Let $N(t, S)$ be the number of times the spawning condition *that results in* jobs of type t is evaluated to *true*, while processing S. Similarly, let $N(S)$ be the number context switches (of any type) while mining S, and N be the total number of context switches during entire execution. We now have,

$$\begin{array}{rcl} N(S) & = & N(task, S) + N(data, S) + N(chunk, S) \\ N & = & \sum_S N(S) \end{array}$$

We now construct the worst case bounds for $N(t, S)$ for each t. While mining S, new *tasks* are created only through adaptive task partitioning. It is performed only after all extensions are produced from S (see Section 4.1). Any subtree can thus produce new tasks *at most* once. We now have,

$$\forall S, N(task, S) \leq 1 : \sum_S N(task, S) \leq \sum_S 1 = |FS| \quad (3)$$

where FS is the set of all frequent subtrees. When a task J spawns jobs onto tree pool, each unexplored tree in $J.i$'s projected database is created as new job. Once the tree pool is processed, it is *guaranteed* that all trees in the projected database are processed for extensions. Thus for any subtree, the switch from task parallel mode to data parallel mode can happen *at most* once.

$$\forall S, N(data, S) \leq 1 : \sum_S N(data, S) \leq |FS| \quad (4)$$

Finally, $N(chunk, S)$ is equal to the number of trees in S's projected database which spawn the chunk-level jobs. From Section 4.3, jobs of type $chunk$ are created only when TP is empty. We can thus infer that $N(chunk, S)$ is always less than the number of cores. If $N(chunk, S) \geq |C|$ then TP can not be empty. Therefore,

$$\forall S, N(chunk, S) \leq |C|-1 : \sum_S N(chunk, S) \leq |FS| * (|C|-1) \quad (5)$$

From Equations 3- 5,

$$\begin{aligned} N &= \sum_S N(S) \\ &= \sum_S N(task, S) + \sum_S N(data, S) + \sum_S N(chunk, S) \\ &\leq |FS| + |FS| + |FS| * (|C| - 1) \\ &\leq |FS| * (|C| + 1) \end{aligned}$$

Thus, the number of context switches per pattern is bounded by a constant, and the total number N is in the order of $|FS|$. However in practice, these numbers are very very small since the algorithm moves to a lower granularity only when the parallelism at current granularity is *completely* exploited. For example, many subtrees would have already been enumerated by the time the first data-parallel job is created i.e., $\sum_S N(data, S) \ll |FS|$.

4.5 Scheduling Service

A key outcome of our efforts in adaptive parallelization is a scheduling service that has been ported to two multicore chips and one SMP system. We believe that such services will be ubiquitous as systems grow more complex and are essential to realize performance commensurate with technology advances. For simplicity, we limit our discussion to the basic interface shown in Algorithm 5. Functions I1 and I2 are basic start and clean-up routines. Jobs in our system are implemented using job descriptors (see above). Once the service is started, I3 specifies the list and the order among different granularities which the application wants to exploit. It also creates different job pools and other data structures used for scheduling. $gOrder$ determines the order in which the job pools are accessed. For each granularity, I4 defines an application handle that is invoked to execute the the job of that granularity. I5 (optionally) registers a synchronization callback handle that is used for jobs whose return flag is set to true. I6 is responsible for scheduling and completing all jobs by performing context switches, if needed (similar to Alg. 4). I7 and I8 are invoked for the creation and execution of jobs. I9 is a check point function used to evaluate whether or not to switch between different granularities.

Algorithm 5 Prototype interface for scheduling service

```
I1  void startService ()
I2  void stopService ()
I3  int register ( int *granularities, int size, int *gOrder )
I4  int bind ( int gran, void (*callback) (void *) )
I5  int finalize ( int gran, void (*sync) (void *) )
I6  void schedule ()
I7  int createJob ( int gran, void *inputs )
I8  int executeJob ( job *j );
I9  bool evaluateForSpawning ( job *j )
```

The way we invoke different routines from the interface is shown in Alg. 4. Different granularities are set up by invoking I3, I4, and I5 in line 2. Lines 11 and 13 call the *sync* from I5 since the jobs at *data* and *chunk* level require coordination. Entire scheduling of jobs i.e., lines 3-15 make up the implementation of I6.

In this article we have specifically employed this service for the task of tree mining but we expect it to be useful for a range of pattern mining tasks (from itemsets to graphs) as well as more broadly for other data-intensive applications. For example, one can easily parallelize famous algorithms like FPGrowth [12] and gSpan [40] using our service (see Section 6). The current implementation is limited to CMPs and SMPs but we are in the process of extending this service for cluster systems comprising of multicore nodes. We also plan to implement this service on top of Intel's TBB and other related libraries for portability, as opposed to current pthreads implementation. As we show later in Section 5.3, our service is capable of producing some useful performance statistics. We leverage this feature in designing a performance monitoring tool that provides *real time* feedback to applications. It can be used in a variety of applications running on CMP architectures.

5. EMPIRICAL EVALUATION

We evaluate our algorithms using two commonly used real-world data sets, Treebank (TB) [6] and Cslogs (CS) [42] – derived from computation linguistics and web usage mining, respectively. The number of trees and the average tree size (in number of nodes) in CS and TB are (59691, 12.94) and (52581, 68.03), respectively. We use a 900 MHz Intel Itanium 2 dual processor system with 4GB RAM, and if more memory is required (typically by extant algorithms), we use a system with 32GB RAM (same processor) instead of relying on virtual memory.

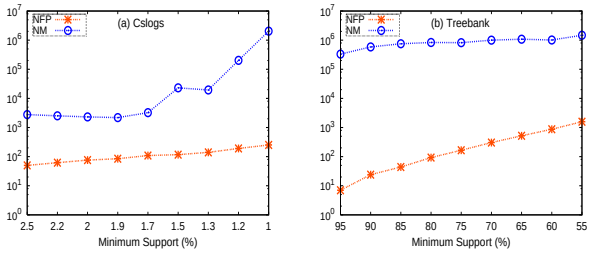

Figure 4: Change in NM and NFP as a function of $minsup$

We consider two data set characteristics which affect the performance – number of frequent patterns NFP (affects the run time), and average number of matches per frequent pattern NM (affects the memory usage). While NFP depends on $minsup$, we find that many frequent patterns found in both datasets have a large number of matches in the data. While the trees in TB possess a very deep recursive structure, the tree nodes in CS exhibit a high variance in their label frequencies. This high variance makes NM to increase at a much faster rate in CS as we decrease the support (see Fig. 4). We will pinpoint the influence of these properties when discussing the relevant experimental results. Hereafter, DS-$minsup$ denotes an experiment where DS is a data set and $minsup$ is the support.

5.1 Sequential Performance

Effect of optimizations: We highlight the benefits from our optimizations in Figure 5 by considering the run time and memory usage of Trips as the baseline. Note that the Y-axis in 5b & 5d is shown in *reverse* direction to indicate the reduction in memory usage. The memory footprint of algorithms is approximated as its resident set size (RSS) obtained from the "top" command. The results shown for each optimization include the benefits from all the other optimizations presented before that. So, $CHUNK$ refers to fully optimized Algorithm 3 (MCT).

Even though the dynamic lists from NOEM decrease the memory consumption of Trips, they add to the run time overhead due to redundant recursions in Alg. 2. In case of TB-$40K$ [7] alone, NOEM slowed down Trips by 3.6 times – due to 10 billion recursions in finding just 413 million subsequences, which include about 289 million false positives (i.e., about 7 out of 10). While LF and DOM streamline the backtracking process to reduce the number of recursions to mere 554 million, SIMUL eliminates all 289 million false

[6] `http://www.cs.washington.edu/research/xmldatasets/`

[7] We chose high supports for TB as the data set is highly associative.

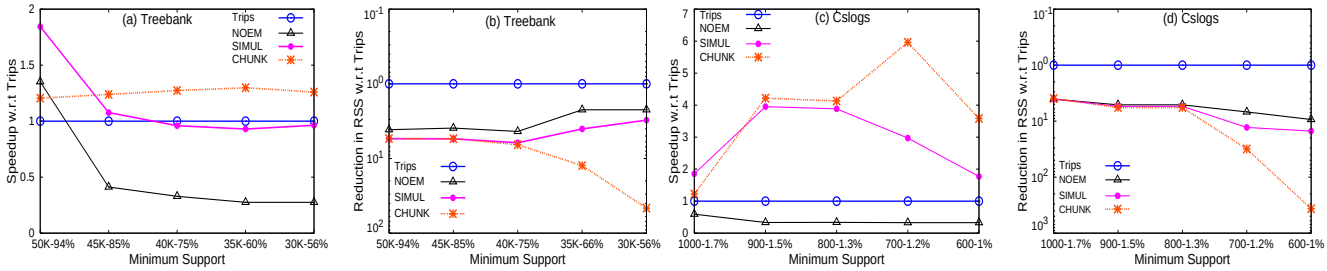

Figure 5: Performance comparison with Trips as the baseline (a&b) Treebank (c&d) Cslogs

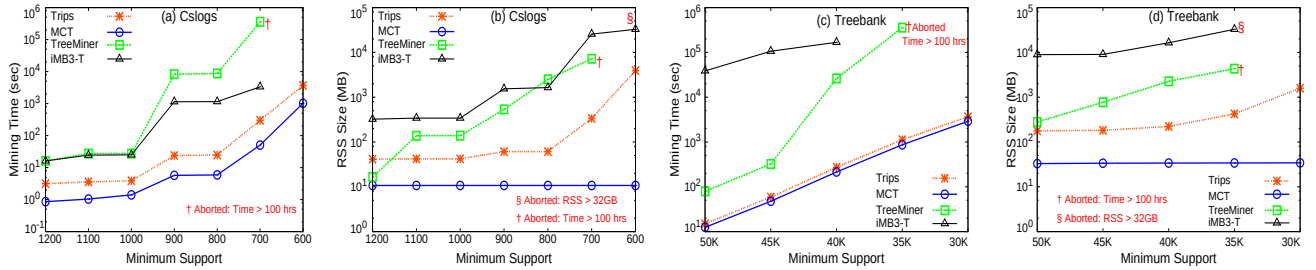

Figure 6: Results on real-world data sets (a&b) Cslogs (c&d) Treebank

positives – giving a 23% run time improvement over Trips. More importantly, these optimizations improve the run time *without affecting* the memory benefits from NOEM. Subsequently, CHUNK (or MCT) by reorganizing the computation, *improves the locality* and *reduces the working sets* resulting in a very good run time and memory performance. When compared to Trips, on $TB-30K$, MCT performs 24% faster and uses 45-$times$ lesser memory. Similarly on CS-600, our optimizations improve the memory usage by 366-*folds* and run time by 3.7-em times.

Comparison with TreeMiner: The performance of TreeMiner is limited by the number and the size of scope-lists, which depend upon the data set properties like NM (see Fig. 4). For example, when a frequent edge in Cslogs is grown into a 6-node pattern, the number of matches increased sharply from 11,339 to 141,574 to 2,337,127 to 35,884,361 to 474,716,009 – resulting in large scope-lists which are later used in expensive joins. Due to such patterns, as the support is changed from 1000 to 800, the memory and run time performance degraded by more than 300 times and 18.5 times, respectively (see Figure 6). In contrast, MCT always maintains a *constant sized* footprint – 10.72MB on Cslogs & 34MB on Treebank – irrespective of the support threshold. Since chunking keeps a *fixed* number of matches in memory at any given point in time, MCT is able to *regulate the memory usage* – a significant result for CMPs where the bandwidth to memory is precious. On CS-700, while TreeMiner ran for more than 100 *hours* with a footprint that is larger than 7GB, MCT took about 50*sec* exhibiting a 7200-fold speedup along with 660-*fold reduction* in memory usage. Even if we factor out the algorithmic benefits from Trips, the benefits from our optimizations are quite significant.

In case of Treebank, the deep recursive structure among trees limits the performance of TreeMiner (see Figures 6c & 6d). As a result, even a small change in support (from $50K$ to $35K$) degrades the performance significantly (by more than *three orders*). On TB-$35K$ alone, MCT exhibits more than 400-*fold speedup* and 120-*fold* smaller memory footprint.

Comparison with iMB3-T: iMB3-T takes a parameter "level of embedding" (L) that controls the type of subtrees that are mined. When L is left unspecified, it mines embedded subtrees – Figure 6 obtained using this setting. Multiple large data structures and apriori-style mining of iMB3-T results in very large memory footprints. Note that, its memory is affected by both NM and NFP, which increase exponentially with the decrease in support (see Fig 4). On CS-700, memory and run time performance of MCT is better than iMB3-T by 66-*times* and 2,300-*times*, respectively. iMB3-T is aborted at CS-600 as its memory usage exceeded 32GB – no corresponding data point in Fig. 6a. It stores the set of all descendants for every frequent node, and hence the deep recursive structure in TB results in very large footprints even at high support values (e.g., 8.5GB at $50K$ support). On TB=$40K$, MCT is 780-*times* faster than iMB3-T while using 480-*times* lesser memory.

5.2 Characterization study for CMP architectures

We now show that our optimizations are suitable for multicores by collecting several hardware performance counters using $PAPI$ toolkit [8]. To this purpose, we run a TB-$45K$ experiment on a system with 1.4GHz Itanium 2 processor and 32GB memory [9].

Analysis of cache performance: We demonstrate the effect of all our optimizations, measured in terms of number of cache misses, in Fig. 7a by taking NOEM in Alg. 2 as the baseline. Tree matching optimizations improve the cache performance by more than 19 times – while LF shrinks R-matrices, DOM and SIMUL reduce the number of data accesses, thereby improving L2 and L3 misses. Added to that, CHUNK *localizes* the computation to higher level caches, and improves the L1 misses of NOEM by a factor of 1,442. A step-by-step effect of various optimizations on run time is shown in Fig. 8d. Overall, *simultaneous matching* and especially *computation chunking* help in achieving very good cache performance.

Analysis of bandwidth pressure: Since all the cores of a CMP system share a single memory bus, memory bandwidth becomes a key factor to application performance. We devise a novel and sim-

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

[9] On-chip caches: 16KB L1-data; 16KB L1-instruction; 256KB L2; and 3MB L3.

ple method to approximate the memory bandwidth by observing the amount of traffic on the front side bus (i.e., off-chip). We first divide the execution time (X-axis in Fig. 7b-d) into small *one msec* slices – a coarse-grained analysis. Then the amount of off-chip traffic during each slice (Y-axis) is approximated to be the product of *L3 line size* and the *number of L3 misses in that slice* (recorded by PAPI).

Figures 7b-d show the variations in off-chip traffic for TreeMiner, Trips, and MCT, respectively. iMB3-T is not considered here due to its poor run time and memory performance. Initial spikes in these figures denote cold L3 misses incurred while bootstrapping (e.g., reading the data set). Frequent accesses to large memory-bound scope-lists result in very high off-chip traffic for TreeMiner. Each cluster of points in Figure 7c denotes the traffic seen while mining a single subtree. The spikes followed by sudden dips indicate the non-uniform nature of computation in Trips. In contrast, the well-structured computation of MCT results in *more uniform and small sized* memory requests. On an average, accesses made by MCT are *well below* 200*KB per msec* whereas the accesses made by TreeMiner and Trips are sized more than 1100KB and 600KB per msec, respectively. This difference is even more while mining the patterns with large number of matches – compare small spikes around 6000 *msec* in Fig. 7d with the large ones around 8000 *msec* in Fig. 7c. From this coarse-grained study it appears that each core in TreeMiner, and to a lesser extent in Trips, aggressively attempts to access main memory (due to embedding lists). For instance, on a dual quad-core system from Section 5.3, we observed a sustained *cumulative* bandwidth of 1.5GB per sec. With 1100KB per msec accesses (i.e., 1GB per sec per core) by TreeMiner, the bandwidth is likely to saturate it is executed on multiple cores. Overall, our optimizations *reduce* the *off-chip traffic* and *its variability*, making them viable for CMPs.

Analysis of working set size: We empirically examined the working sets maintained by different algorithms using Cachegrind [10]. We monitored the change in L1 miss rate by varying the L1 size from 2KB to 256KB (L2 size and its associativity is fixed). We found that L1 miss rate of MCT reduced sharply between 8KB and 16KB and stayed constant for L1 size $>$ 16KB. This suggests that the *working set size* is *between* 8*KB and* 16*KB*. As shown in Table 1, other algorithms maintain relatively larger working sets. This is an encouraging result with respect to CMPs as the amount of cache available for each core is likely to be small [14].

5.3 Parallel Performance

We evaluated our parallel algorithms on a dual quad-core E5345 Xeon processor system [11] – see Figure 8a. Our adaptive load balancing strategies achieve near-linear speedups up to 7.85-folds on CS and 7.43-folds on TB, when all 8 cores are used. We also considered a 16-node SMP system [12] to test the scalability of our techniques. As shown in Figure 8b, the speedup continues to increase with the number of processors, giving a 15.5-fold speedup with all 16 processors. Load balance achieved by individual strategies for TB-$45K$ is demonstrated in Figure 8c.

An important observation from Figure 8b is that *the need for fine-grained strategies increases as one increases the number of processors*. For CS-600, the performance of hybrid strategy (HyP) reaches its plateau at 12 processors ("CS-600 Hybrid" in Fig. 8b) due to a 6-node pattern that has up to 33 million matches in a single database tree, whose mining took about 45*sec*. Amdahl's law suggests that HyP can never perform better than 45*sec* since it is limited by the job of mining a single tree. Thereafter the efficiency can only be improved by employing more fine-grained strategies such as the one in Section 4.3. Similarly for TB-$35K$, the speedup from HyP saturates at 16 processors.

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

[11] 6GB RAM, 8MB shared L2, and 1333 MHz bus speed.

[12] A SGI Altix 350 system with 16 1.4GHz Itanium 2 processors and 32GB memory.

Cores (|C|)	1	2	4	8	16
N_t	0	4	7	26	48
N_d	0	2	2	10	11
N_c	0	0	0	9	19

Table 2: Cost analysis on TB-$35K$, $\|FS\|$=451

on TB-$45K$	Cores	1	2	4	6	8
Trips	EqP	1.00	1.61	1.94	1.95	2.01
	AdP	1.00	1.77	2.23	2.25	2.30
TreeMiner	EqP	1.00	1.61	1.94	1.95	2.01
	AdP	1.00	1.77	2.23	2.25	2.30

Table 3: Parallelization of Trips and TreeMiner

The average number of context switches taken over 10 runs of TB-$35K$ is shown in Table 2. For a given granularity g, $\sum_S N(g,S)$ is denoted as N_g in the table. When $|C|$=1, there are no context switches as the work is not shared at any level. As $|C|$ increases, we see more and more context switches at fine-grain level reflecting the fact that our strategies *adaptively also automatically* exploit the parallelism at all levels of granularity. It is worth noting that these numbers are much lower than their theoretical upper bounds from Section 4.4: N_t=48 $\ll$ $|FS|$=451; N_d=11 $\ll$ 451; and N_c=19 $\ll$ 451*(|C|-1), where |C| is number of cores. Similar results on CS-600 can be found in our technical report [29].

Note that the performance numbers in Table 2 are directly obtained from our service. We designed an interesting performance monitoring tool by leveraging the capability of our service to produce such useful numbers and our light-weight mechanism to approximate achieved memory bandwidth (see Section 5.2). Such a tool not only is capable of providing real time feedback to applications but is also useful to understand the performance characteristics of many applications on CMPs.

The results in Fig. 8 are obtained using a global job pool. However, our service can handle distributed or hierarchical job pools. Further, we expect the contention overhead due to global job pools to be very small as the locking on CMPs is likely to be cheap [13].

Parallel speedups of Trips and TreeMiner using our task-level methods are shown in Table 3. The rationale for these results is as follows. Inherent dependency structure in embedding lists and scope-lists make it difficult to apply more fine-grained strategies to Trips and TreeMiner, respectively. Since these lists are maintained on per-pattern basis, data partitioning methods like HyP, which construct the lists in parallel incur significant synchronization overhead. Further, excessive use of dynamic data structures in TreeMiner serializes the heap accesses, affecting the parallel efficiency – as $|C|$ is changed from 1 to 8, the system time (from the "time" command) increased by more than 4 times. Techniques like *memory pooling* are *ineffective* here as these data structures grow arbitrarily in size. These results re-emphasize the following mantra for good parallel efficiency: *reduce the memory footprints*; *reduce the use of dynamic data structures*; and *reorganize the computation* so that more fine-grained strategies can be applied.

We next discuss the broader outcomes of our study, directions for future research and highlights key results.

[13] `http://download.intel.com/technology/architecture/sma.pdf`

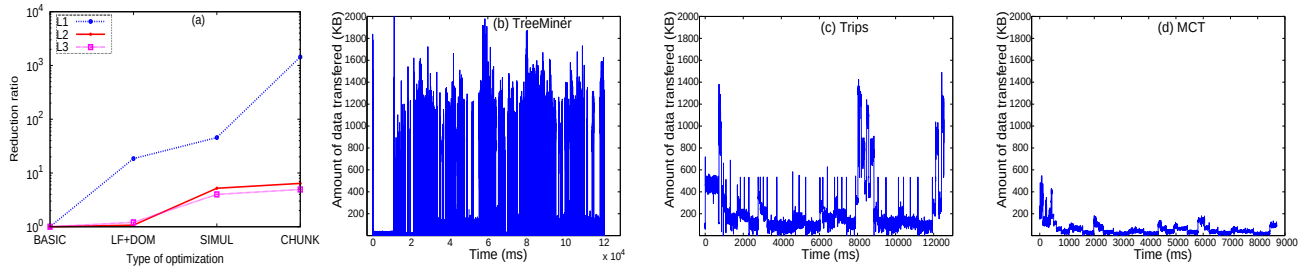

Figure 7: Characterization of optimizations

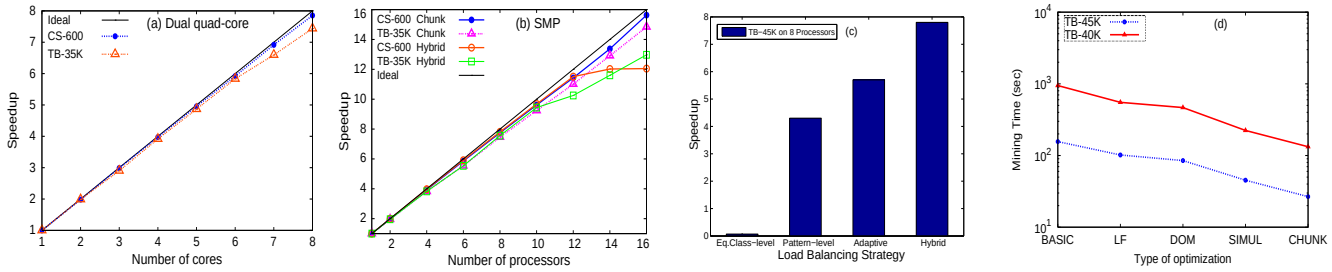

Figure 8: (a, b, c) Parallel performance; (d) Effect of optimizations

6. DISCUSSION

Memory optimizations: Improving locality (spatial or temporal) continues to be important, but in addition, bandwidth must also be considered when designing data-intensive algorithms for emerging CMPs. The traditional trade-off between time and space, and its implications for parallelism need to be examined carefully in this light. All our memory optimizations target the above challenges. Each optimization may not amount to a significant improvement on its own but the specific orchestration applied when combining them yields significant savings – L1 misses reduced by up to $1,442$ times, memory footprints reduced by a factor of 366, bandwidth pressure reduced significantly by making uniform small-sized accesses to main memory, and overall run time reduced by a factor of four on sequential execution.

These optimizations have a broader applicability in many domains. We briefly mention some of the important ones. NOEM essentially improves the mining by computing the required matches *on-demand* instead of *storing* extra state. Such a technique is evidently useful in mining other types of patterns such as graphs, sequences. Techniques similar to NOEM are also useful in searching bioinformatic databases [36] and XML repositories [47].

The tree matching optimizations (LF, DOM, SIMUL), though appear to be specific to Prüfer sequences, have a general purpose utility. They can easily be adapted to algorithms which rely on a depth first encoding. They can also be used to reduce the overhead in other dynamic programming based approaches – mining time series [5]; establishing maximal matchings between glycan structures [2]; code generation techniques [1]; (multiple) sequence alignment [17]; and computing consensus and agreement of phylogenetic trees [27]. This list by no means is an exhaustive one.

Computation chunking captures the general notion of breaking the computation into smaller pieces so that they can be handled efficiently. Such an approach have a general purpose utility in database query processing [19] and also in mining other structures such as graphs, DAGs, induced subtrees, and sequences. In gSpan [40], instead of finding children for each occurrence of a subgraph separately, one can group a set of occurrences and find the one-edge growths collectively. When applied to induced subtree mining, these optimizations exhibit a speedup of 15-folds against FreqT [29]. They can also potentially be leveraged for answering reachability queries on directed graphs – a direction of research that we are actively pursuing.

Applications seldom realize *peak* memory bandwidth numbers quoted in product specifications [37]. These numbers often assume that the references are uniformly distributed across the memory system, and avoid conflicts for banks and front side bus – a rare case, in practice. It is thus very important to look at *achieved* bandwidth, especially for CMPs where the memory bus is shared among all cores. The method from Section 5.2, though an approximate one, provides an easy and quick way to study the memory behavior of algorithms at individual *core level*. We believe that this lightweight mechanism to measure the bandwidth is widely applicable to several other data mining and database applications [19].

Out-of-core performance: The out of core performance of our optimizations has not been evaluated in this article. This is primarily because the focus of this work is on multicore performance and also due to limits on space. However, since the reader may be interested in this question we would like to note that our algorithms, primarily due to the memory optimizations, easily translate to a reasonably good out-of-core implementations. In fact a straightforward realization results in an out-of-core implementation that is about six times slower than the in-core algorithm for the evaluations described in this paper. Once the dataset exceeds the limits of main memory the performance of the in-core implementation degrades rapidly whereas the out-of-core implementation sees a much more graceful degradation. An interesting observation here is that computation chunking has an even larger role to play in our out-of-core realization (by a factor anywhere from 5 to 10) since chunk-wise processing localizes the computation to run in memory. We are currently exploring more sophisticated out-of-core algorithms that leverage hash-based data placement, in a manner similar to our previous work on out-of-core frequent pattern mining [7].

Parallel algorithms and scheduling service: With regards to

task scheduling, algorithms that can adapt and mold are essential to achieve performance commensurate with the number of cores in emerging CMP systems. Coarse-grained strategies are usually not sufficient since systemic, parametric and data-driven constraints make the workload estimation a challenging task. In such scenarios the ability of an algorithm to adaptively modulate between coarse grained and fine grained strategies is critical to parallel efficiency. In fact how much an algorithm can adapt essentially dictates when the performance plateau is reached, as we observed in our study. Our adaptive strategy demonstrated near-perfect parallel efficiency on both a recent CMP and a modern SMP system.

A key outcome here beyond the specific tree mining algorithm is the realization of a general purpose scheduling service that supports the development of adaptive and moldable algorithms for database and mining tasks. For instance, one can parallelize a graph mining algorithm like gSpan [40] by simply defining the task descriptors and appropriate job spawning conditions. Rest of the details like job scheduling, synchronization, and thread management are *transparently* taken care by our service. This service is easily applicable to many other pattern mining algorithms because they all employ a pattern-growth approach and traverse the search space in depth first order. They are also applicable to other data mining tasks like classification using decision trees [45].

Acknowledgments

This work is supported in part by grants from National Science Foundation NGS-CNS-0406386, CAREER-IIS-0347662, RI-CNS-0403342, and CCF-0702587.

7. REFERENCES

[1] A. Aho, M. Ganapathi, and S. Tjiang. Code Generation Using Tree Matching and Dynamic Programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, 1989.

[2] K. Aoki, A. Yamaguchi, Y. Okuno, T. Akutsu, N. Ueda, M. Kanehisa, and H. Mamitsuka. Efficient Tree-Matching Methods for Accurate Carbohydrate Database Queries. *Genome Informatics Series*, pages 134–143, 2003.

[3] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Satamoto, and S. Arikawa. Efficient Substructure Discovery from Large Semi-structured Data. *In SDM*, pages 158–174, 2002.

[4] I. Baxter, A. Yahin, L. Moura, M. SantAnna, and L. Bier. Clone Detection Using Abstract Syntax Trees. *In ICSM*, pages 368–377, 1998.

[5] D. Berndt and J. Clifford. Finding patterns in time series: a dynamic programming approach. *Advances in knowledge discovery and data mining*, pages 229–248, 1996.

[6] G. Buehrer, S. Parthasarathy, and A. Ghoting. Adaptive parallel graph mining for CMP architectures. *In ICDM*, pages 97–106, 2006.

[7] G. Buehrer, S. Parthasarathy, and A. Ghoting. Out-of-core frequent pattern mining on a commodity PC. *In KDD*, pages 86–95, 2006.

[8] E. Charniak. Tree-bank grammars. *In AAAI*, pages 1031–1036, 1996.

[9] Y. Chi, R. Muntz, S. Nijssen, and N. Kok. Frequent Subtree Mining-An Overview. *Fundamenta Informaticae*, 66(1):161–198, 2005.

[10] Y. Chi, Y. Yang, Y. Xia, and R. Muntz. CMTreeMiner: Mining Both Closed and Maximal Frequent Subtrees. *In PAKDD*, pages 63–73, 2004.

[11] H.H. Gan, S. Pasquali, T. Schlick. Exploring the repertoire of RNA secondary motifs using graph theory; implications for RNA design. *In Nucleic acids research*, 31(11):2926–2943, 2003.

[12] J. Han, J. Pei, and Y. Yin. Mining Frequent Patterns without Candidate Generation. *In SIGMOD*, pages 1–12, 2000.

[13] K. Hashimoto, I. Takigawa, M. Shiga, M. Kanehisa, and H. Mamitsuka. Mining significant tree patterns in carbohydrate sugar chains. *In Bioinformatics*, 24(16):i167-73, 2008.

[14] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction. *In IEEE Micro*, pages 81–92, 2003.

[15] S.Y. Le, and *et al*. RNA secondary structures: comparison and determination of frequently recurring substructures by consensus. *In Bioinformatics*, 5(3):205–210, 1989.

[16] H. Leung and H. Ting. An optimal algorithm for global termination detection in shared-memory asynchronous multiprocessor systems. *In TPDS*, 8(5):538–543, 1997.

[17] S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *In Journal of Molecular Biology*, 48(3):443–453, 1970.

[18] S. Nijssen and J. Kok. Efficient Discovery of Frequent Unordered Trees. *In MGTS*, pages 55–64, 2003.

[19] L. Qiao, V. Raman, F. Reiss, P. Haas, and G. Lohman. Main-memory scan sharing for multi-core cpus. *In VLDB*, pages 610–621, 2008.

[20] C.F. Olson. Parallel algorithms for hierarchical clustering. *In Parallel Computing*, 21(8):1313–1325, 1995.

[21] P. Rao and B. Moon. PRIX: indexing and querying XML using prufer sequences. *In ICDE*, pages 288–299, 2004.

[22] U. Ruckert and S. Kramer. Frequent Free Tree Discovery in Graph Data. *In ACM SAC*, pages 564–570, 2004.

[23] B. Saha and *et al*. Enabling scalability and performance in a large scale CMP environment. *In EuroSys*, pages 73–86, 2007.

[24] B.A. Shapiro and K. Zhang. Comparing multiple RNA secondary structures using tree comparisons. *In Bioinformatics*, 6(4):309–318, 1990.

[25] D. Shasha and J. Zhang. Unordered tree mining with applications to phylogeny. *In ICDE*, pages 708–719, 2004.

[26] J. Srivastava, R. Cooley, M. Deshpande, and P. Tan. Web usage mining: discovery and applications of usage patterns from Web data. *ACM SIGKDD Explorations Newsletter*, 1(2):12–23, 2000.

[27] M. Steel and T. Warnow. Tree Theorems: Computing the Maximum Agreement Subtree. *Information Processing Letters*, 48:77–82, 1993.

[28] H. Tan, T. Dillon, F. Hadzic, E. Chang, and L. Feng. IMB3-Miner: Mining Induced/Embedded Subtrees by Constraining the Level of Embedding. *In PAKDD*, pages 450–461, 2006.

[29] S. Tatikonda and S. Parthasarathy. Mining Tree Structured Data on Multicores: An Adaptive Architecture Conscious Approach. *Technical Report OSU-CISRC-TR18*, The Ohio State University, 2007(updated October 2008).

[30] S. Tatikonda, S. Parthasarathy, and M. Goyder. LCS-TRIM: Dynamic Programming meets XML Indexing and Querying. *In VLDB*, pages 63–74, 2007.

[31] S. Tatikonda, S. Parthasarathy, and T. Kurc. TRIPS and TIDES: new algorithms for tree mining. *In CIKM*, pages 455–464, 2006.

[32] A. Termier, M. Rousset, M. Sebag, K. Ohara, T. Washio, and H. Motoda. Efficient Mining of High Branching Factor Attribute Trees. *In ICDM*, pages 785–788, 2005.

[33] A. Termier, M. C. Rousset, and M. Sebag. DRYADE: A New Approach for Discovering Closed Frequent Trees in Heterogeneous Tree Databases. *In ICDM*, pages 543–546, 2004.

[34] R. Wagner and M. Fischer. The String-to-String Correction Problem. *In JACM*, 21(1):168–173, 1974.

[35] C. Wang, M. Hong, J. Pei, H. Zhou, W. Wang, and B. Shi. Efficient Pattern-Growth Methods for Frequent Tree Pattern Mining. *In PAKDD*, pages 441–451, 2004.

[36] J. Wang, H. Shan, D. Shasha, and W. Piel. TreeRank: A Similarity Measure for Nearest Neighbor Searching in Phylogenetic Databases. *In SSDBM*, pages 171–180, 2003.

[37] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *In SC*, pages 1–12, 2007.

[38] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. *In PLDI*, pages 30–44, 1991.

[39] Y. Xiao and J. Yao. Efficient data mining for maximal frequent subtrees. *In ICDM*, pages 379–386, 2003.

[40] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. *In ICDM*, pages 721–724, 2002.

[41] L. Yang, M. Lee, and W. Hsu. Finding hot query patterns over an XQuery stream. *In VLDB*, 13(4):318–332, 2004.

[42] M. Zaki. Efficiently Mining Frequent Trees in a Forest: Algorithms and Applications. *In TKDE*, 17(8):1021–1035, 2005.

[43] M. Zaki. Parallel and Distributed Association Mining: A Survey. *In IEEE Concurrency*, pages 14–25, 1999.

[44] M. Zaki and C. Aggarwal. XRules: an effective structural classifier for XML data. *In KDD*, pages 316–325, 2003.

[45] M. Zaki, C. Ho, and R. Agrawal. Parallel classification for data mining on shared-memory multiprocessors. *In ICDE*, pages 198–205, 1999.

[46] K. Zhang. Computing similarity between RNA secondary structures. *In IEEE Joint Symposia on Intelligence and Systems*, pages 126–132, 1998.

[47] P. Zezula, G. Amato, F. Debole, and F. Rabitti. Tree signatures for XML querying and navigation. *In XSym*, pages 149–163, 2003.