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 also shown to reduce the pressure on the front-side bus, an important consideration, when executing data-intensive algorithms on emerging large-scale multicore systems. 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 makes load balancing a significant challenge. To address this challenge, we present a methodology that adaptively modulates from coarse grained task partitioning to the finest granularity of partitioning needed, as dictated at runtime by the input parameters and the dataset properties, while minimizing overhead costs. 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. Examples abound ranging from analysis and management of XML repositories [29] to phylogenetic analysis [28], from web mining [28] to analyzing glycan structures [8], from analyzing linguistic data [3] to examining parse trees [2]. In most of these instances, the essential problem may be abstracted to the one that of discovering frequent patterns from a database of rooted ordered trees, the focus of this article.

VLDB '08, August 24-30, 2008, Auckland, New Zealand

Motivated primarily by power and energy considerations recent advancements in computer architecture have seen the advent of chip multiprocessor (or multi-core) systems (CMPs). Such systems are becoming extremely common-place, and the general trend in processor development has been from single-core to many-core: from dual-, quad-, eight-core chips to the ones with tens of cores [1]. For such architectures, 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 [9, 11]. Equally important, it becomes imperative to identify scalable and efficient parallel algorithms to deliver performance commensurate with the number of cores (processing elements) 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.

It is our contention that extant tree mining algorithms will require significant changes to meet these challenges. The rationale is as follows. First, a majority of these algorithms employ special data structures to store extra state with which they avoid repeated executions of expensive subtree isomorphism checks, a key operation in such algorithms [1, 5, 12, 14, 17, 20, 21, 24, 28]. These data structures house redundant information[17, 28], may rely on pointer-based designs[24], and are often persistent across the entire execution[17]. Such mechanisms that trade off space for time are acceptable and often effective on unicore systems with large memory capacities but expected to be inefficient on multicore architectures where the premium on off-chip memory accesses is expected to be even higher. Moreover, parallel instantiations of such algorithms will require shared access to such data structures and often dictate housing additional redundant information thereby reducing the overall efficiency. Second, even if the first issue can be effectively resolved through appropriate memory conscious designs, one still needs to design an effective parallelization strategy accounting for workload skew. This is particularly challenging, since it is often difficult to estimate the workload of a task, even at runtime, for such applications since the influence of dataset characteristics and input parameters must be accounted for and accurately estimated. Additionally, as several recent researchers have pointed out there is a need to expose and subsequently exploit fine-grained parallelism on such architectures [15].

As an effort in this direction, we propose several memory conscious optimizations to improve the locality, limit the working set size, and alleviate the bandwidth pressure on the front side bus of such algorithms. While these optimizations are presented and evaluated in the context of Trips[20] a state-of-the-art sub-tree mining

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

algorithm, the key ideas are quite generic and can often be easily incorporated to other algorithms targeting the mining and management of structured data. Our Memory Conscious Trips (MCT) reduces the memory usage of Trips by up to 366-times and improve the run time by up to 4 times. When compared to other extant algorithms, we demonstrate over *three orders* improvement in memory usage, and more than *two orders* improvement in run time.

We then empirically demonstrate that these optimizations while necessary are not sufficient for efficient parallelization on multicores, primarily due to parametric and data-driven factors which makes load balancing a significant challenge. To address this challenge, we present a methodology that adaptively modulates from coarse grained task partitioning to the finest granularity of partitioning needed, as dictated at runtime by the input parameters and the dataset properties, while minimizing overhead costs. The resulting algorithm achieves near perfect parallel efficiency on up to 16 processing elements 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.

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.*

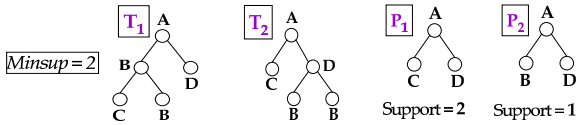

Figure 1: Example database and patterns

The minimum support ($minsup$) can either be expressed 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* or *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-based support is 1 whereas its occurrence-based support is 2. In this article, we use the transaction-based definition. Figure 1 shows two example database trees (T_1, T_2) and patterns (P_1, P_2). P_1 has one embedding in both T_1 and T_2, whereas P_2 occurs only in T_1 (with 2 embeddings). If $minsup$=2 then we have a single *frequent* subtree, P_1. A variant of this problem mines for *induced*, as opposed to *embedded*, subtrees [2].

There are two important phases in any tree mining algorithm, *candidate generation* and *support counting*. The first phase generates the candidate subtrees which are then evaluated for their frequency in the second phase. A key challenge in the first phase is to efficiently traverse the search space. The challenge in support counting is to quickly check for the existence of a subtree in a given database tree (i.e., subtree isomorphism). We employ a pattern-growth method where a frequent subtree S is repeatedly *grown* with new edges to yield new candidate subtrees. The process of attaching an edge is called as *point growth* and the edge itself is called as an *extension*. An equivalence class of S (denoted as [S]) contains the set of all frequent subtrees generated from S through *one or more* point growths. For instance, if S is a single node (say v) then [S] contains all frequent subtrees whose root is v.

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

	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 (see Section 7)

[2] Maximum memory footprint observed in *all* our experiments in Section 7

Table 1: Characterization of Tree Mining Algorithms

Related Work: A majority of existing tree mining methods maintain *embedding lists* (EL) in order to speedup the mining process. The set of all matches of a frequent subtree S is stored in its EL so that the embeddings of subtrees from its equivalence class [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 [28]. It generates new candidates by *joining* different scope-lists. Scope-lists however suffer from *redundancy* and *occupies a lot of memory* (see Table 1), especially when the number of overlapping embeddings is high – a common case in most real-world data sets. All-in-all large memory also lead to expensive run time performance for low support queries.

Asai *et al* proposed *FreqT* that operates in a similar manner to TreeMiner but mines *only induced* subtrees. It maintains the occurrences of the right-most leaf (as an embedding list) and uses them in generating new candidates. Induced subtree mining is much easier than embedded subtree mining since parent-child relations are simpler than ancestor-descendant relations.

Tan *et al* proposed *iMB3* that relies on occurrence-based support and mines both induced and embedded subtrees [17]. They recently developed a similar algorithm that uses a transaction-based support, which hereinafter, is referred to as *iMB3-T*. Both iMB3 and iMB3-T employ several data structures: *dictionary* to represent the data; *descendant list* to track *all* descendants of a frequent node; and *occurrence list* to store the embeddings. The memory consumption of iMB3-T is typically very high since the first two potentially large data structures are maintained across the entire execution. Occurrence lists further increase the memory consumption to unacceptable levels, even at moderate values of $minsup$ (see Table 1).

Wang *et al* proposed *Chopper* and *XSpanner* [24]. Chopper recasts the problem of subtree mining into sequence mining. However, its performance is often hindered by large number of false positive subsequences. XSpanner, on the other hand, mines for subtrees by performing recursive projections. As we note in a previous study, its performance suffers from complicated projections and pointer-chasing and typically is outperformed by TreeMiner [20].

Researchers have also proposed algorithms for mining closed and maximal frequent subtrees, which significantly reduce the output size. These algorithms also suffer from similar performance issues. For example, CMTreeMiner [5] and PathJoin [26] can mine only induced subtrees. Extending these ideas for mining embedded subtrees is not trivial. Termier *et al* proposed algorithms for mining closed embedded subtrees [22] but they assume that no two sibling nodes can have the same label – an unrealistic assumption especially in case of real-world data sets. In addition, they suffer from memory issues since they also employ data structures similar to descendant lists in iMB3-T. There exist several other algorithms which differ in the type of subtrees that they mine [12, 14, 16]. Please refer to the survey paper by Chi *et al* for more details [4].

In this work we present our memory optimizations in the context of Trips, an algorithm that relies on a sequence-based representation of tree structured data [20]. Of all the algorithms discussed thus far it has the smallest memory footprint and working set size

(see Table 1 for a comparison among leading contenders), and as we show in our empirical results it is also the most efficient. While these numbers appear quite reasonable for Trips, at lower support values, they can still be much too large for emerging multi-core systems. We next briefly describe the Trips algorithm.

Trips [20]: Trips encodes each database tree T as two sequences: NPS_T – Numbered Prüfer Sequence; and LS_T – Label Sequence. Trips constructs these sequences iteratively based on post-order traversal numbers (PON) associated with every node in T. In every iteration, it removes the node (say, v) with the smallest PON. The label of v is appended to LS_T, and the PON of v's parent is added to NPS_T. An example Prüfer sequence is shown Figure 2.

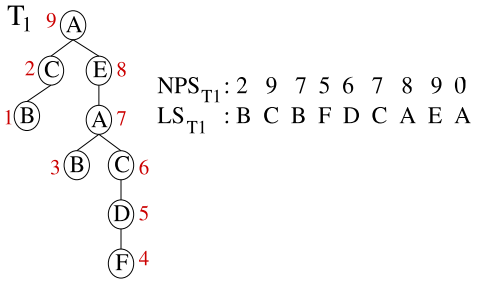

Figure 2: Example tree and its Prüfer sequence

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 $newtidlist$ // to indicate $newpat$ 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, $newtidlist$)

The complete Trips algorithm is shown in Algorithm 1. It first transforms the given database trees into sequences (D) and determines the set of all frequent nodes (F_1). For each $f \in F_1$, *mineTrees* is called to mine subtrees from its equivalence class [f].

mineTrees is a recursive procedure with three parameters: a pattern pat, an extension point (lab, pos), and a projected database $tidlist$. An extension (lab, pos) of pat defines a new subtree that is obtained by attaching a node with label lab to a node in pat whose PON is equal to pos (=$newpat$ in line 1). Each extension point thus uniquely identifies a subtree that is grown from a given pattern. The *projected database* (PD) $tidlist$ contains the list of trees in which pat has at least one embedding.

EL(T) initially contains the set of all embeddings of pat in tree T. Lines 3-5 transform this list into EL for $newpat$ by appending the locations in T at which (lab, pos) matches – equivalent of finding isomorphisms of $newpat$ in $tidlist$. PD for $newpat$ is simultaneously built by adding T to $newtidlist$ in line 6. When *mineTrees* returns, the newly appended entries associated with (lab, pos) are deleted so that only the matches of pat remain in EL(T).

Lines 8-12 scan each tree in $newtidlist$ to determine extensions (i.e., point growths) of $newpat$. Every node in T is evaluated against every embedding of $newpat$, given by EL(T) (lines 9-10). All valid extensions are then hashed into H, which maintains their frequencies (lines 11-12). Lines 13-14 determine the *frequent* extensions and they are recursively mined by invoking *mineTrees*. Note that, each extension in H denotes a unique subtree that is obtained by adding a node to $newpat$.

3. CHALLENGES

We next briefly outline some of the key challenges one needs to address in order to get effective performance on emerging multi-core systems for such applications.

1. Memory footprint size: The control of memory footprint size is an important consideration for both sequential and parallel algorithms. Excessive memory usage may force OS to rely on virtual memory thereby slowing down the application. It may also increase the bus contention and consequently the number of stalls – likely to be aggravated on CMPs since cores usually share a common bus to the main memory. As Table 1 illustrates, the memory footprint for state-of-the-art algorithms is quite high.

2. Locality of reference and working set sizes: In emerging multicore systems, since a sizable portion of the chip's real estate is occupied by the cores themselves, the amount of space available for on-chip caches and local stores is expected to be somewhat limited[9, 11]. Algorithms designed for CMPs must maintain small-sized working sets to deliver good performance – a side effect of having small caches. The working set is the amount of data that is actively used by the program during a particular phase of computation. If working sets are large then the off-chip traffic increases due to constant data swapping between cache and memory. Again referring to Table 1, it is illustrative to see that at a moderate support value the working sets for the three algorithms listed range from 64KB to 256KB. With lower supports, and on systems with many cores the problem will be more severe. Efforts to improve locality in pattern mining algorithms to address this challenge is *non-trivial* since spatial locality is deterred by pointer-based data structures, and temporal locality is hindered by huge search space.

3. Load Balance: Good scalability can only be achieved by keeping the processors busy for as much time as possible by effectively partitioning and distributing the work among processors. It is a challenging task since it is difficult to estimate the time to mine a given pattern. The mining time depends not only on the workload and data set characteristics but also on the input parameters.

4. MEMORY OPTIMIZATIONS

Though embedding lists (EL) are designed to trade space for improved execution time, they can grow arbitrarily in size resulting in poor memory and run time performance, especially at low support values. As an illustration, consider the embedding lists in Trips (see Algorithm 1). Assume a worst case scenario of a *chain tree* (i.e., a path) of size n, where every node has the same label (say, A). For a single node pattern (i.e., A), EL would contain exactly $\binom{n}{1}$=n entries. When it is extended with a node to produce new subtree A-A (an edge), 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. For example, in case of Cslogs (a real-world data set – see Section 7), when a 3-node pattern is grown into a 6-node subtree, the number of matches sharply increased from $141,574$ to $474,716,009$ – resulting in a proportional increase in the memory footprint size. We address these limitations by designing a series of memory optimizations.

The architecture of our Memory Conscious Trips (MCT) is shown in Figure 3. The database trees (D) are first transformed into Prüfer sequences (T(D)) – see Section 2. Tree pruning phase prunes the infrequent nodes from T(D) to produce T′(D) (see Section 4.1). Both T′(D) and the set of frequent nodes $F1$ are fed to the mining block that contains three phases: on-the-fly embedding lists OEL (see Section 4.2), candidate generation CG, and support counting SC. Instead of storing the complete embedding list, CG invokes OEL to compute the matches on-demand (see Section 4.3). CG and SC then operate on the matches to produce frequent extensions (point growths). Generated extensions are fed back into the mining block for producing larger patterns. The computation in OEL is reorganized so that only a fixed number of matches (called as *chunk*) are generated at any given time. Instead of finding all matches at once, CG always requests OEL for a fixed number of matches (called as *chunk* – see Section 4.3). Once the processing of one chunk is complete, it requests OEL for the next chunk of matches.

4.1 Tree Pruning and Recoding (PRUNE)

This avoids superfluous computations by eliminating those parts of the tree which do not help in finding frequent subtrees. It draws inspiration from a well-known technique in itemset mining [7]. It prunes the database trees from infrequent nodes and recodes the remaining nodes. Pruning alters the label sequence and recoding changes the numbered Prüfer sequence (see Algorithm 2). Once the size of pruned tree is computed (lines 3-5), labels are recoded from root to leaves so that a node's NFA is known at the time of pruning it. NFAs are maintained in map array, and they are used to build the new NPS (line 4). $newLab$ stores the recoded labels which are updated whenever a frequent node or an infrequent root node is found. The time complexity of both pruning and recoding steps is $O(n)$, where n is the number of nodes in the tree.

Algorithm 2 Tree pruning and recoding algorithm

```
Input: Database D, Recoded labels newLab = φ
 1: for each tree T in D do
 2:     n ← size of T;   T_recoded ← null;   count ← 0
 3:     for each node v ∈ T do
 4:         if v is the root or v is frequent then
 5:             increment count
 6:     ind ← count − 1
 7:     for i from n to 1 do
 8:         v ← i^th node in the Prüfer sequence of T
 9:         if v is root then
10:             add v to newLab, if not present
11:             add v to T_recoded with recoded label
12:             map[n] ← count
13:         else if v is frequent then
14:             u ← (newLab(v.label), map[v.parent])
15:             add u to T_recoded
16:             map[i] ← ind;   ind ← ind - 1;
17:         else
18:             map[i] ← map[v.parent]
```

4.2 On-the-fly Embedding Lists (NOEM)

As mentioned, a majority of algorithms leverage embedding lists (EL) to trade space for time. Instead of storing EL explicitly, we adopt the following strategy: dynamically *construct* the embedding list; *use* it; and then *de-allocate* it. In graph-theoretic terms, the problem of dynamic list construction is equivalent to the one that asks for the set of all (embedded) subtree isomorphisms of a given pattern in the database. Inspired by our recent research in XML indexing that converts XML data into sequences and subsequently makes use of dynamic programming to determine sequence matches [19], we employ a similar approach here. Note that, this optimization affects only the lines 3-6 of Algorithm 1 – *correctness* of the overall algorithm is *still intact*.

Let P=(LS_P, NPS_P) be the subtree whose embeddings need to be found in a database tree T=(LS_T, NPS_T). Also, let m and n be the size of P and T, respectively. The subtree matching problem is recasted into a much simpler problem of subsequence matching by leveraging the following property.

PROPERTY 4.1. *If a pattern P is a subtree of a tree T then the label sequence of pattern LS_P is a subsequence of the label sequence of tree LS_T.*

From Property 4.1, being a subsequence is a *necessary but not sufficient* condition for subtree isomorphism. We also note that, LS_P is a subsequence of LS_T if and only if LS_P is the longest common subsequence (LCS) of LS_P and LS_T. In order to construct the dynamic embedding lists, we first check whether or not LS_P is a subsequence of LS_T by computing their LCS (step 1). If yes, we proceed to enumerate all subsequence matches in LS_T (step 2) and these matches are processed further to determine exact subtree matches (step 3). We now present these three steps, in detail.

Step 1 - *Subsequence Checking*: As mentioned, we check if LS_P is a subsequence of LS_T or not by computing their LCS length. LCS is traditionally computed using a dynamic programming approach [23]. Such an approach constructs a matrix (say R) with LCS lengths for all possible prefix combinations of input strings i.e., LS_P and LS_T (see Equation 1). If |LCS|=$R[m,n]$ is not equal to $|LS_P|$=m then we conclude that P is *not* a subtree of T. An example pattern and its corresponding R-matrix is shown in Figure 3b.

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

Step 2 - *Subsequence Matching*: If LS_P is a subsequence of LS_T then we enumerate all subsequence matches of LS_P in LS_T by backtracking from $R[m,n]$ to $R[1,1]$ (lines 6-11 in Algorithm 3). Whenever the labels of both P and T match, the match location is recorded in SM and the match length L is incremented (lines 5-7). Since backtracking is done in a reverse order, the matches are established from *right-to-left*. Each resulting subsequence match SM is of the form (i_1,...,i_m), where $LS_P[k]$=$LS_T[i_k]$ for $1 \leq k \leq m$ (see Figure 3b for an example).

Step 3 - *Structure Matching*: Since Property 4.1 is only a necessary condition, Step 2 can potentially generate false positive matches. We filter them in this step by matching the structure of P with the structure that is formed by SM. We establish a map ($strMap$) by mapping parent-child relations in P with ancestor-descendant relations in T. More formally,

i) Given: P and a subsequence match (i_1,...,i_m)

ii) Map the root node directly by setting $strMap[m]$=i_m.

iii) For k = m-1...1, check whether $strMap[NPS_P[k]]$ is either

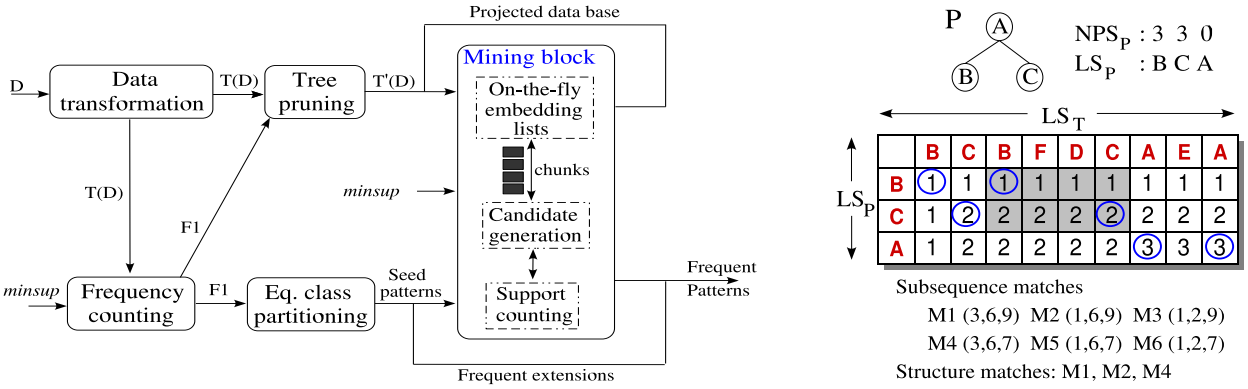

Figure 3: (a) Framework of our algorithm MCT (b) Dynamic list construction: R-matrix for P and T_1 of Figure 2

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**
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 ($p_i, t_j - 1, L$)

equal to $NPS_T[i_k]$ or is a nearest mapped ancestor of $NPS_T[i_k]$. This ensures that the parent of k^{th} node in P is mapped to an ancestor of i_k^{th} node in T. Note that the structure match is also established from right-to-left (i.e., in a root-to-leaf order) so that $u \in P$ is processed only after the structure match for its parent is determined. The complete mapping $strMap$ is then added to the dynamically constructed embedding list.

Example: In Figure 3b, 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., $strMap$[3]=9. The structure agreement check at k=2 (i_k=2) will pass because $strMap$[3]=$NPS_T[i_k]$. However at k=1 (i_k=1), $strMap$[3] is not equal to not the nearest mapped ancestor of i_1 in T. At this point, the structure agreement check fails and we declare that M3 is a false positive. Similarly, for $M5$ and $M6$, the structure agreement check fails at k=1 and k=2, respectively.

The tree matching algorithm in Alg. 3 suffers from two issues: recursion overhead; and the overhead from processing false positive subsequence matches. We now present optimizations which are built on top of this basic algorithm to alleviate these two issues.

Label Filtering (LF): It alleviates the recursion overhead in Alg. 3 by eliminating columns which are not useful in subsequence matching. Let $\exists k$ such that $LS_T[k] \notin LS_P$. Since the entries $R[*, k]$ simply carry forward the LCS values from $R[*, k-1]$ to $R[*, k+1]$, recursions made on them do not help in building the subsequence match – Hence they can be safely deleted from R. LF not only *reduces the redundant computations* but also *shrinks R matrices* making them to fit in few cache lines. In Figure 3b, the columns for D, E, and F can be filtered.

Dominant Match Processing (DOM) LF relies on the distribution of P's labels over the nodes of T. If every label in T occurs in P then LF has no effect on the performance. DOM enforces further restrictions to streamline the backtracking process. DOM strengthens the benefits from LF by limiting the processing to only a selected few entries in R. Let $R[i, j]$ and $R[k, l]$ be two cells at which the LCS length is incremented (condition 2 in Equation 1). Also, say that $\nexists\ x$ such that $1 \leq j < x < l \leq n$, and $LS_P[k] = LS_T[x]$. Such entries are referred to as *dominant* matches. The recursions on cells between $R[i, j]$ and $R[k, l]$ simply carry the LCS length from $R[i, j]$ to $R[k, l]$ without improving the subsequence match. Backtracking from $R[k, l]$ can directly jump to $R[i, j]$ *without* going through all the intermediate cells. The dominant matches are encircled in Figure 3b. For example, $R[1, 3]$ and $R[2, 6]$ are dominant, and recursions on all the shaded cells can be eliminated without affecting the correctness of the algorithm. By restricting recursions to dominant matches, the recursion overhead can be reduced significantly.

Simultaneous Matching (SIMUL) Both LF and DOM do not address the problem of overhead due to false positive subsequences which are fed into the structure matching phase. We note that both subsequence and structure matching phases operate on the sequence from right-to-left. They can therefore be performed simultaneously instead of evaluating the structure after a complete subsequence match is generated. We perform the structure match at position k as soon as the subsequence match at that position is established. By embedding structural constraints into subsequence matching, SIMUL detects the false positives as early as possible and *never* generates them completely.

4.3 Computation Chunking (CHUNK)

Since the size of EL is proportional to the number of matches, the dynamic embedding lists can also be exponential, in a 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 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 num-

ber 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. In our empirical study, we define chunks to contain 10 matches.

Though chunking and tiling are seemingly similar, there are few differences. *First*, while tiling (in a classical sense [25]) groups a set of data items, chunking groups a set of computations which is applied onto a single data item (here, T). We thus refer to these chunks as *computation chunks*. *Second*, tiling improves the cache performance by dividing the data (i.e., T) into parts such that each part fits in the cache where as chunking improves the performance by reducing the number of accesses on T (without dividing it). Unlike tiling, it *does not depend on any hardware parameters* such as cache size making it a *cache oblivious* algorithm as opposed to a cache-conscious algorithm [6].

Algorithm 4 Fully optimized Trips

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 ($p_i - 1$, $t_j - 1$, $L - 1$)

Our complete Memory Conscious Trips (MCT) is shown as Algorithm 4. Since it always keeps a fixed number of matches in memory, MCT maintains a *constant-sized* memory footprint throughout the execution. Further, chunk-level processing helps in *localizing* the computation to higher level caches thereby improving the locality and working sets.

Complexity analysis: Like any other pattern mining algorithm, MCT belongs to *#P complexity class* since it has to count and enumerate all frequent subtrees. The procedure *mineTrees* in Algorithm 4 is invoked exactly once for every frequent pattern (pat+e) that is discovered. For a given S and T (of size m and n, respectively), the maximum number of recursions on *processR* ($c_{m,n}$) can be approximated to the following [19]:

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

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

Our optimizations are general purpose and can be applied to other algorithms. The tree matching algorithm of NOEM is applicable to sequences used in TreeMiner. However unlike Alg. 4, TreeMiner relies on the scope-lists without making any explicit database scans. Therefore, one can not directly apply NOEM and other subsequent optimizations without revamping the entire TreeMiner algorithm by avoiding scope-lists. All our optimizations can be incorporated into iMB3-T with suitable modifications. The *dictionary* strucutre in iMB3-T stores the tree nodes in pre-order. By making simple modifications to the data structure, one can apply our tree matching algorithm in Section 4.2 and the three tree matching optimizations to iMB3-T. Note that this will only eliminate the large *Occurrence lists* maintained by iMB3-T. Other potentially large *dictionary* and *descendant list* data structures can not be avoided unless the entire algorithm is rearchitected.

5. INDUCED SUBTREE MINING

Induced subtrees maintain parent-child relationships whereas embedded subtrees preserve ancestor-descendant relationships. Our optimizations can be tuned to mine induced subtrees by making some simple modifications to our algorithms. While searching for new extensions in line 11 of Algorithm 1, we only need to consider v's parent instead of evaluating all of v's ancestors. Once we apply PRUNE optimization (see Section 4.1), children of a given node may actually correspond to descendant nodes in the original tree. We therefore need to distinguish between *induced* children and *embedded* children. In other words, each edge in the pruned tree has to be annotated to indicate whether it represents a parent-child or ancestor-descendant relation in the original data base tree. Finally, structure agreement checks at line 14 of Algorithm 4 have to consider only the parent node. More precisely, structure agreement check at position k succeeds only if $LS_P[k]$=$LS_T[i_k]$ and $strMap[NPS_P[k]]$=$NPS_T[i_k]$.

6. ADAPTIVE PARALLELIZATION

We now consider the parallelization of MCT for emerging CMP architectures. For the record we would like to note that directly parallelizing the Trips algorithm is the first approach we considered. However, we quickly abandoned this approach as the maintenance of embedding lists led to a large memory footprint resulting in significant contention overhead and pressure on the front-side bus. Moreover, the inherent dependency structure of the lists pose difficulties in sharing them, which lead to coarse grained work partitioning resulting in poor load balance. Essentially, *parallelization without identifying the memory-conscious optimizations*, presented in the previous section, *is extremely inefficient*.

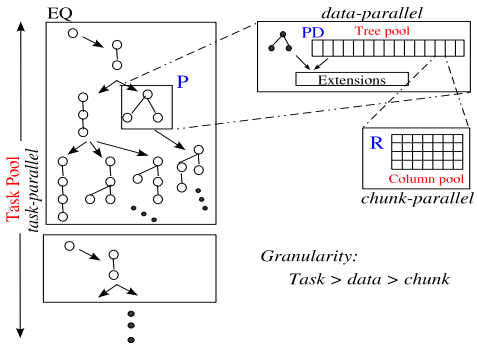

Figure 4: Schematic of different job granularities

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 different threads. Such a strategy helps in obtaining good parallel efficiency by keeping all processors busy for as much time as possible. Each core C_i in the CMP system runs a single instantiation (i.e., a thread) of our parallel algorithm. Therefore, throughout this section, 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.

As Leung *et al* [10] pointed out, if the threads are allowed to share the work asynchronously then detecting a global termination would become non-trivial – since the jobs could be shared while a termination detection algorithm is being executed. Instead, we implement a simple *lock-based* termination detection algorithm that is driven by the amount of work that is available in the system. Whenever a thread C_i finds JP to be empty, it votes for termination and joins a *thread pool* (TP). Before joining TP, C_i detaches itself (i.e., blocks itself) from execution. Though a busy-wait style approach is applicable here, it will be extremely inefficient. Each thread monitors TP at pre-set points during its run time to check if it can share some of its work. If TP is not empty then the thread *may* choose to fork off some new jobs onto JP. At which point, the thread that shared the jobs signals (i.e., unblocks) the threads on TP to indicate the availability of newly generated work. Unblocked threads then proceed to execute the new jobs from JP. The mining process *terminates* when *all* threads vote for termination. We implemented TP using simple locks (akin to semaphores) and condition variables. Note that this algorithm can easily be extended to detect termination in cases where multiple job pools are maintained based on different thread groups (e.g., distributed and hierarchical job pools). Such a design allows job pools to act as implicit channels for communication between running and waiting threads.

For better load balance, we allow threads to operate in three different execution modes: *task-parallel*, *data-parallel*, and *chunk-parallel*. Each mode determines the type and granularity of the work that is being shared among threads in that mode. For a simpler design, we used different job pools for different execution modes: *task pool* (JP_T), *tree pool* (JP_R), and *column pool* (JP_C), respectively – see Figure 4 [3]. Shared access to these pools is protected using simple locks. Individual jobs are uniquely identified by *job descriptors*. Each job descriptor J is a 6-tuple as shown below.

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

The job type $J.t$ determines the remaining entries of the descriptor. A thread starts with the input parameters $J.i$, applies the function $J.f$ to produce an output $J.o$. The control is then returned to the job that created J if the return flag $J.r$ is set to *true*. A condition $J.c$ is evaluated at pre-set points in order to determine whether or not to spawn new jobs from J. The job type $J.t$ also determines the type of new jobs that J can spawn. If $J.t = task$ then J can either create new tasks onto JP_T or a single job of type $data$. A job in JP_R can only create jobs of type $chunk$. Finally, jobs of type $chunk$ can not create any new jobs i.e., $\forall J \in JP_C$, $J.c = false$. The granularity of jobs in JP_T is more than that of in JP_R, which in turn is greater than the granularity of jobs in JP_C. The pseudocode that integrates different execution modes and our termination detection algorithm is shown as Algorithm 5. Such a design *adaptively adjusts the granularity* by switching between the execution modes.

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

Algorithm 5 Parallel Tree Mining

```
1: while true do
2:     if JP_T is empty then
3:         if JP_R is empty then
4:             if JP_C is empty then
5:                 vote for termination
6:                 block itself from execution
7:                 if { all threads voted } break
8:             else
9:                 process columns from JP_C // chunk-parallel
10:        else
11:            operate on trees from JP_R // data-parallel
12:    else
13:        mine a task from JP_T // task-parallel
```

6.1 Task-parallel mode

In this mode, jobs are shared at a granularity of *tasks* through JP_T. A task is always associated with a subtree S, and is defined as the piece of work that mines (i.e., finding extensions) S and also a *subset* of patterns from its equivalence class $[S]$.

For each job $J \in JP_T$, $J.i$ is a subtree and $J.o$ is the set of all frequent subtrees generated (from $J.i$) by invoking $J.f$ ($mineTrees$ procedure from Alg. 4). Further, $J.r$ is always set to *false* in this mode. J continues to execute until one of the following holds: (i) the spawning condition $J.c$ is evaluated to *true*; or (ii) all the subtrees from the class $[J.i]$ are enumerated. $J.c$ is repeatedly evaluated at pre-set points in Algorithm 4. We identify three different strategies to partition the search space into tasks.

Equivalence class -level task partitioning (EqP): In this strategy, threads operate at the highest level of granularity.

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

$J.i$ is an element of F1 in Algorithm 1 and corresponds to a coarsest partition of the search space – denoted as EQ in Figure 4. Since the spawning condition is set to $false$, the output contains the set of all frequent subtrees whose root is $J.i$ (= $[J.i]$). This strategy performs very well when the output sizes $|J.o|$'s $\forall J \in JP_T$ are in the same order. However, most real-world data sets are highly skewed and the variance in $|J.o|$'s is considerably high. Therefore, this strategy often results in very poor load balance.

Pattern -level task partitioning (PaP): EqP exploits the parallelism between equivalence classes whereas this strategy exploits the parallelism between individual patterns. Each job corresponds to the task of mining a single pattern (P in Figure 4).

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

JP_T is initialized with the seed patterns from F1. If |F1| $< |C|$ then we initialize with the extensions from seed patterns (i.e., frequent edges). One can continue to mine in levels until the number jobs are sufficiently greater than the number of cores. $J.o$ contains all extensions produced from S. Since $J.c$ is a tautology, every extension in $J.o$ is used to create a new task on JP_T. For better efficiency, along with the subtree, one can also supply its projected database (i.e., $newtidlist$) as part of $J.i$.

Aggressive job sharing in this strategy often results in poor memory management and high computation overhead. Further, this technique also suffers from poor locality since it does not guarantee that the subtrees are mined at the place where they are generated.

Adaptive task partitioning (AdP): Both EqP and PaP strategies operate at two extreme levels of granularity. While EqP is

too conservative, PaP is too aggressive in job sharing. This strategy operates in some middle-ground by *adaptively modulating* the granularity. Here, JP_T is initialized in a similar manner to PaP.

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

|Ext| is the number of frequent extensions derived from $J.i$ that are *yet to be* processed. J spawns new tasks into JP_T only when the thread pool is not empty and there exists a few unprocessed extensions which can be shared with idle threads in TP. Note that the condition $TP \neq \Phi$ implies that the job pool is empty. This is a stringent requirement because new jobs are not spawned until JP_T becomes empty. Instead, we can choose to spawn new tasks when the number of jobs in JP_T ($|JP_T|$) falls below a pre-defined threshold, $thresh$. $J.c$ is evaluated before processing each extension i.e., between lines D-E of Algorithm 4.

Adaptive task partitioning *achieves better load balance* when compared to EqP and PaP because of its dynamic modulation of task granularity. In addition, it *exploits the locality* by mining the extensions on the processor that created them, whenever possible.

6.2 Data-parallel mode

Task-parallel strategies achieve load balance by migrating tasks among different threads. They implicitly rely on an assumption that all patterns are of equal complexity. However due to inherent skew in most of the real-world data sets, the variance in mining times of different patterns is typically very high. The resulting load imbalance can be alleviated by parallelizing the job of mining a single subtree S. The key step in mining S is the scan on S's projected database $newtidlist$ (PD_S) – lines 8-12 in Algorithm 1 (PD in Figure 4).

In a naive *data partitioning* strategy, a job is defined as the work associated with each tree in PD_S. In other words,

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

Note that JP_R (unlike JP_T) is defined in the context of a subtree that is currently being mined. The function $J.f$ finds embeddings of S in the tree $J.i$ and produces the corresponding extensions of S (i.e., $J.o$). $J.r$ is set to true so that the thread that spawned jobs into JP_R can perform a reduction operation to combine the partial set of extensions discovered by each job in JP_R. This approach creates a tree pool for every pattern and hence it can suffer from heavy synchronization overhead. Instead, we design a strategy that dynamically switches from task-parallel mode to data-parallel mode when task-level parallelism can not be exploited any further.

Hybrid work partitioning (HyP): Let a core C_i is processing a task-level job $J \in JP_T$ with $J.i$=S. If C_i finds any idle threads *while* finding extensions from S then C_i switches to data-parallel mode and forks off new jobs onto JP_R. Once JP_R is processed by co-operating threads, the control returns to the place where J spawned the data-parallel jobs. At which point, J performs the reduction operation on the partial sets of extensions. Note that, J *may* now proceed (if needed) to create new tasks according to *AdP* strategy. 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.

$$\forall J \in JP_T,$$
$$J.c = \left\{ \begin{array}{ll} spawn\ tasks\ onto\ JP_T, & \text{if } |JP_T| < thresh \\ spawn\ jobs\ onto\ JP_R, & \text{if } TP \neq \Phi \wedge \\ & J.f\ is\ not\ complete \end{array} \right\}$$

First condition is evaluated between the lines D-E of Alg. 4 (similar to AdP) whereas the second condition is evaluated between the lines A-B. We can further strengthen the second condition by adding an additional criterion, "*mining J.i is expensive*". A simple predictive estimate for cost of mining $J.i$ can be obtained based on its size, its support (already known – line 14 in Alg. 1), and the number of matches found so far.

To further improve the load balance, we employ a heuristic that is used in classical job scheduling, where the jobs are sorted in the decreasing order of their processing time. Similarly, before creating the tree pool, we sort the trees in PD_S in the decreasing order of their size. This is because the time to mine extensions from a tree is likely to be proportional to its size (line 9 in Alg. 1).

6.3 Chunk-parallel mode

In this mode, we move to a much finer level of granularity and parallelize the job of mining a single database tree – which includes finding embeddings and associated extensions from the tree. We allow multiple threads to operate on a single R-matrix so that the embeddings are enumerated in parallel. A job in JP_R switches into this mode based on following condition:

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

Alternatively, 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. It is evaluated between lines 13-14 of Alg. 4.

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

6.4 Cost analysis

The benefits from our multi-level adaptive work sharing approach will be undermined if the cost of context switching between different execution modes is very high. We now evaluate the number of context switches by analyzing the job spawning conditions. Note that the evaluation of all spawning conditions is a constant time operation. 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 the entire mining process. 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 build worst case bounds on $N(t, S)$ for each t. While mining S, new jobs of type $task$ are created only through adaptive task partitioning, which is performed only after all the extensions are produced from S (see Section 6.1). Hence a given subtree can produce new tasks *at most* once. We now have,

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

In a case where a task-level job J spawns jobs onto tree pool, a job is created for each unexplored tree in $J.i$'s projected database. Whenever the control returns back to J, it is *guaranteed* that all trees in the projected database are processed for extensions. Thus for any subtree, the context switch from task parallel mode to data parallel mode can happen *at most* once.

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

Finally, $N(chunk, S)$ is equal to the number of trees in S's projected database which spawn the chunk-level jobs. From Section 6.3, the spawning condition simply checks whether or not the thread pool is empty. We can thus infer that $N(chunk, S)$ can never be equal to or exceed the number of cores, $|C|$. If $N(chunk, S) \geq |C|$ then TP can not be empty. Therefore,

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

From Equations 3- 5,

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

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., $\sum_S N(data, S) \ll |FS|$.

6.5 Scheduling Service

A key outcome of our efforts in adaptive parallelization is a scheduling service that has currently 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 algorithmic performance commensurate with technology advances. For expository simplicity, we limit our discussion to the basic interface shown in Algorithm 6. Functions I1 and I2 are basic start and clean-up routines. Jobs in our system are implemented using job descriptors, as described in previous sections. 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 internal data structures used for scheduling. The specified $gOrder$ determines the order in which different 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. 5). 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 6 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 )
```

An example pseudocode using different interface routines is shown as Alg. 7. Once the service is started one registers the two levels of granularity and binds specific functions to be executed at each level. Assuming that the lowest granularity requires synchronization, a call to *finalize* is made. Subsequently jobs at the coarse grained granularity are spawned off. funcA includes calls to I9 to determine if a more fine-grained strategy is desired.

Algorithm 7 Sample pseudocode using the service interface

```
startService()
int grans[ ] = {A, B}, order[ ] = {0, 1};
register ( grans, 2, order );
bind ( A, funcA ) ;
bind ( B, funcB ) ;
finalize ( B, syncB );
for each Eq in F1 do
    createJob ( A, Eq )
schedule()
stopService()
```

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. The current prototype of the service is limited to general purpose multicore systems and SMPs but we are in the process of extending this service for cluster systems comprising multicore nodes.

7. EMPIRICAL EVALUATION

We now evaluate the sequential and parallel algorithms on two commonly used real-world data sets, Treebank (TB) [4] and Cslogs (CS) [28] – derived from the domains of 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. For sequential algorithms, we use a 900 MHz Intel Itanium 2 dual processor system with 4GB RAM. Whenever the memory requirement is more than 4GB, we use a system with the same processor but with 32GB RAM instead of relying on virtual memory. We evaluate our parallel methods on both a CMP system and also on a large SMP system.

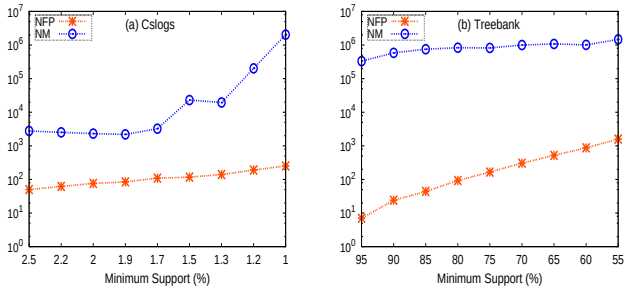

Figure 5: Change in NM and NFP as a function of $minsup$

We consider two data set characteristics which significantly 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). Both Cslogs and Treebank data sets have a small number of frequent patterns and each frequent pattern, on average, has a large number of embeddings in the data (see Figure 5). While the trees in Treeabank posses a very deep recursive structure, the tree nodes in Cslogs exhibit a high variance in their label frequencies. As a result, NM increases at a much faster rate in Cslogs as one decreases the minimum support. Hereafter,

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

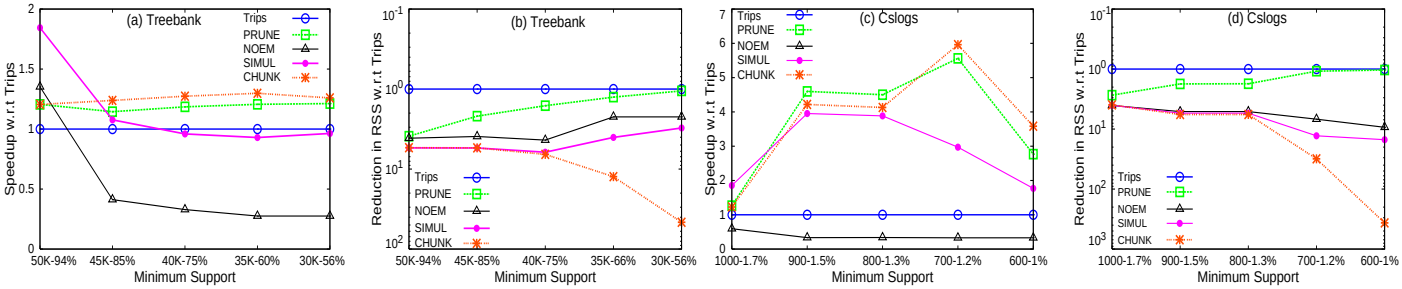

Figure 6: Performance comparison with Trips as the baseline (a&b) Treebank (c&d) Cslogs

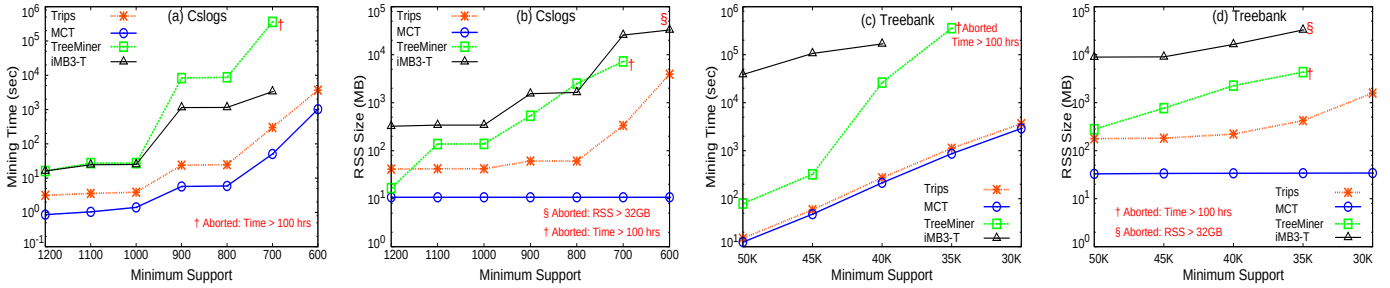

Figure 7: Results on real-world data sets (a&b) Cslogs (c&d) Treebank

we refer to a mining experiment as DS-$minsup$ where DS is a data set and $minsup$ is the support.

7.1 Sequential Performance

7.2 Comparison with Trips

In order to demonstrate the benefits from our optimizations, we consider the run time and memory usage of Trips as the baseline (see Figure 6). Note that the Y-axis in 6b & 6d is shown in *reverse* direction to indicate the reduction in memory usage. The memory footprint of algorithms is approximated by the resident set size (RSS) obtained from the "top" command. The results shown for each optimization *includes* the benefits from all the optimizations presented before that. For instance, $NOEM$ refers to the algorithm obtained by adding both $PRUNE$ and $NOEM$ to Trips; and $CHUNK$ denotes the complete optimized Memory Conscious Tree Miner (MCT).

Consider a single experiment TB-$40K$ [5] (see Figures 6a & b). First, PRUNE cut down the memory footprint of Trips almost by half, from 222MB to 138MB. By constructing the required state dynamically, NOEM further brought down the RSS by 2.7 times to 51MB. However, the computational overhead in Algorithm 3 slowed down the mining process by 3.6 times – 10 billion recursions for finding 413 million subsequences. Subsequently, LF and DOM alleviated this overhead and reduced the number of recursions to a mere 554 million. Approximately 289 million out of 413 million subsequences (i.e., about 7 out of 10) were false positives, which were eliminated by SIMUL thereby improving the run time by 23%. Finally, CHUNK reduced the memory usage of Trips by a factor of 6.5. Similarly for TB-$30K$, our optimizations improved the memory footprint of Trips *by* 45 *times* (1.5GB → 34MB), and its run time *by* 24%.

Our optimizations resulted in much higher benefits on Cslogs due to large variance among the frequencies of different nodes (see Figures 6c & d). PRUNE, NOEM, and CHUNK (on an average) reduced the memory usage by a factor of 1.7, 9, and 82, respectively. In addition, the tree matching optimizations improved the run time of Trips, on an average, by 4 times. For CS-600 alone, our optimizations brought an improvement of 366-*folds* in memory usage, and a speedup of 3.7-*folds* in run time.

To summarize, the benefits from PRUNE are marginal especially at low support thresholds. Though NOEM improves the memory usage by constructing the lists on demand, it suffers from huge computational overhead. Such an overhead is alleviated by our tree matching optimizations LF, DOM, and SIMUL. Computation chunking completely eliminates the need for explicit maintenance of embedding lists, and as a result, *significantly reduces* the memory usage of Trips. Further, the well-structured chunk-by-chunk processing *improves the locality*, *reduces the working sets*, and results in improved mining times.

7.3 Comparison with TreeMiner

Figure 7 shows the comparison between MCT and TreeMiner. While mining CS or TB, TreeMiner typically stores large scope-lists since the average NM is very high in these data sets. For instance, when a particular frequent edge in Cslogs is grown into a 6-node pattern, number of matches increased sharply from $11,339 \rightarrow 141,574 \rightarrow 2,337,127 \rightarrow 35,884,361 \rightarrow 474,716,009$. Resulting large lists are then used in expensive join operations. Therefore, not only the memory usage but also the run time performance is significantly affected. When $minsup$ is decreased from 1000 to 800, the run time of TreeMiner has increased by more than 300 times (from 27*sec* to $8,748$*sec*), and memory has gone up by 18.5 times (from 135MB to $2,505$MB). In contrast, the run time increase in MCT is very small, and more interestingly, *without* any change in memory consumption.

MCT maintains a *constant* sized memory footprint at *all* support levels – 10.72MB on Cslogs & 34MB on Treebank. Since chunking keeps a *fixed* number of matches in memory at any given point in time, our optimizations were able to *regulate the memory requirements* of the algorithm – a significant result for CMP architectures where the bandwidth to memory is precious. At $minsup$=700, TreeMiner was aborted after running for 100 *hours*, at which point the memory footprint was more than 7GB. On the other hand, MCT took about 50*sec* (a 7200-fold speedup) and maintained a 10.72MB memory footprint (more than 660-*fold reduction*). Even if we fac-

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

tor out the algorithmic benefits from Trips [20], the benefits from our optimizations are quite significant.

Similar conclusions can be drawn from the results on TB (see Figures 7c & 7d). The performance of TreeMiner is again mediocre as a result of deep recursive structure present among trees in Treebank. When $minsup$ is reduced from $50K$ to $35K$, the run time of TreeMiner increased exponentially by more than *three orders* of magnitude. At TB-$35K$, it was aborted after running for 100 *hours* (memory usage then was $>$ 4GB) whereas MCT spent 866*sec* and used 34MB of memory in mining all frequent subtrees – more than 400-*fold speedup* and 120-*fold reduction* in memory footprint.

7.4 Comparison with iMB3-T

The comparison between MCT and iMB3-T is shown in Figure 7. iMB3-T takes a parameter "level of embedding" (L) which controls the type of subtrees that are mined. It mines embedded subtrees when L is left unspecified – Figure 7 is obtained using this setting. The memory usage of iMB3-T is usually very high due to multiple potentially large data structures and also because of its apriori-style mining (embedding lists for many subtrees have to be maintained). In other words, both NM and NFP govern the memory consumption of iMB3-T, which increases exponentially with the reduction in support threshold. As the support is changed from 1000 to 700 on CS, its memory usage is increased quickly from 340MB to 25GB with the proportional 130-fold slowdown in run time. At CS-700, MCT exhibited more than $2,300$-*fold reduction* in memory usage along with 66-*fold speedup*. Note that, iMB3-T was aborted at CS-600 as its footprint exceeded 32GB, and hence there is no corresponding data point in Figure 7a. On Treebank data set, its memory footprints are large even at high support values (e.g., 8.5GB at $50K$ support). This is because of deep recursive tree structures in TB – iMB3-T stores the set of all descendants for every node. At $minsup$=40K, MCT is 780-*times* faster and used 480-*times* less memory. At the support of $35K$, it was again aborted as the memory consumption exceeded 32GB.

7.5 Comparison with FreqT

We now evaluate the induced subtree mining algorithm presented in Section **??** against both FreqT [6] and iMB3-T (L=1) (see Figure 8). In terms of scale, the run times in Figure 8 are much smaller than that of in Figure 7. This is because the induced subtrees are much easier to mine when compared to embedded subtrees. When compared to FreqT, MCT demonstrated up to 10-fold improvement in memory usage, and more than 15-fold speedup in run time. Since FreqT stores only the location of right most leaf nodes, its memory footprint is not very high. However, such simple strategies can not be extended for mining embedded subtrees.

In contrast, iMB3-T, even at very high supports, keeps large memory footprints and exhibits very poor run time performance. For example at $minsup$=50%, the memory usage iMB3-T was more than 8GB ($>>$ 62MB of MCT), and the mining time was more than $7,500$*sec* ($>>$ 64*sec* of MCT). The memory footprint of iMB3-T on Cslogs is approximately 30 times bigger than that of MCT. Similarly, the run time of iMB3-T is very high when compare to the mining times of both MCT and FreqT.

7.6 Characterization study for CMP architectures

We now present a detailed characterization of our optimizations to show their applicability for CMP systems. These results are obtained by conducting a TB-$45K$ experiment on a system with 1.4GHz Itanium 2 processor and 32GB memory [7], and by collecting several hardware performance counters using $PAPI$ toolkit [8].

Analysis of cache performance: By considering Algorithm 3 as the baseline, we compare the reduction in number of cache misses due to various optimizations in Figure 9a. Note that this comparison is not to show that NOEM performs poorly but to demonstrate the effect of different optimizations, which are built on top of NOEM. All optimizations reduce the amount of work done and they all show some improvement in $L1$ misses. The tree matching optimizations improve the cache performance by 19 times: R-matrices are shrunk by LF; the number of accesses to R are reduced by DOM; L2 and L3 performance is improved by SIMUL. In addition, chunking helps in localizing the computation to higher level caches and improves the L1 and L3 performance by a factor of $1,442$ and 5, respectively. A step-by-step improvement in run time due to various optimizations is shown in Figure 10d. Overall, *tree pruning and recoding*, *simultaneous matching*, and *computation chunking* help in achieving very good cache performance.

Analysis of bandwidth pressure Figures 9b-d present the variation in off-chip traffic during the execution of TreeMiner, Trips, and MCT respectively iMB3-T is not considered here because of its large memory usage and poor run time. We divided the execution time (X-axis) into small *one msec* slices – a coarse-grained analysis. The off-chip bandwidth during each slice (Y-axis) is approximated to (L3 line size) * (number of L3 misses recorded by PAPI in that slice). With this experiment, though a coarse-grained one, we want to bring out the effect of our optimizations on the memory access pattern.

Initial spikes in these figures denote the cold L3 misses incurred while bootstrapping (e.g., reading the data set). Evidently, the off-chip traffic of TreeMiner is more due to frequent accesses to large memory-bound scope-lists. Each cluster of points in Figure 9c corresponds to the amount of data transferred while mining a single subtree. The spikes followed by sudden dips indicates the non-uniform nature of computation in Trips. MCT in contrast always performs well-structured computations thereby resulting 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 (similar to the one at 6000 *msec*) in Figure 9d with corresponding large spikes (similar to the one around 8000 *msec*) in Figure 9c. 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, we observed a sustained *cumulative* bandwidth of 1.5GB per sec for *all* cores of a dual quad-core system (see Section 7.7). With 1100KB per msec accesses (i.e., 1GB per sec per core), the bandwidth is likely to saturate when TreeMiner is executed on multiple cores. Overall, our optimizations *reduce* the overall *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 [9]. We monitored the change in L1 miss rate as L1 size was varied from 2KB to 256KB (L2 size and its associativity was fixed). We found that L1 miss rate of MCT has 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 Ta-

[6] `http://chasen.org/~taku/software/freqt/`

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

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

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

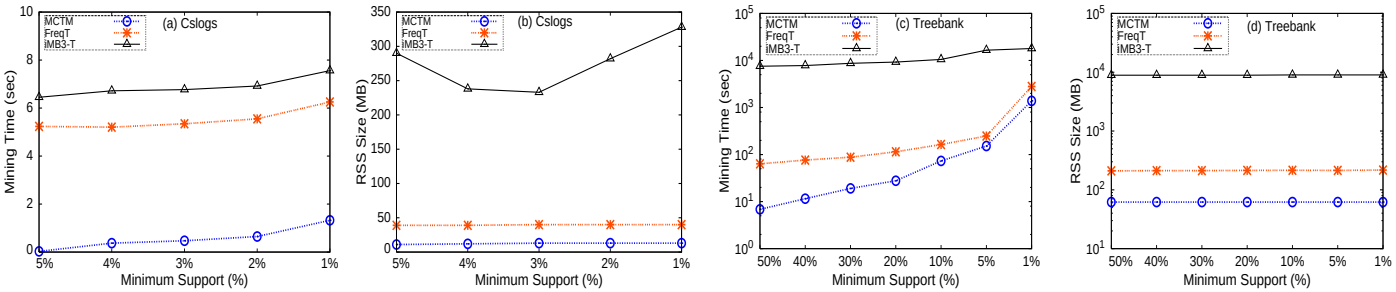

Figure 8: Induced subtree mining (a&b) Cslogs (c&d) Treebank

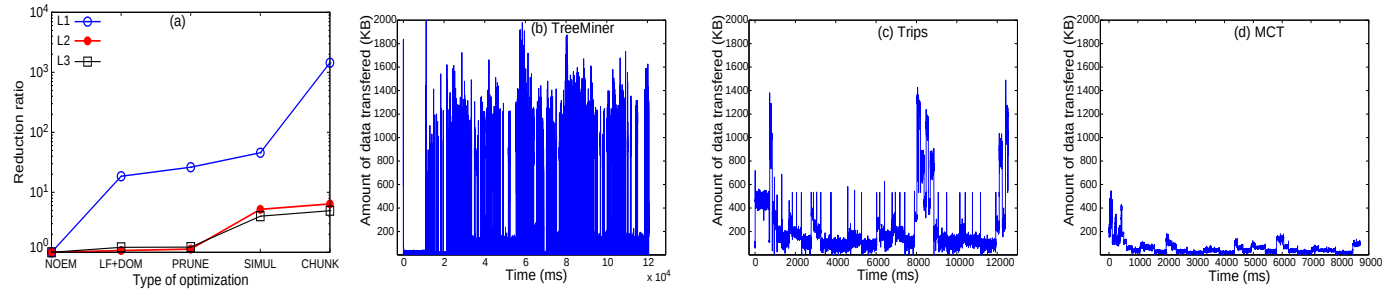

Figure 9: Characterization of optimizations

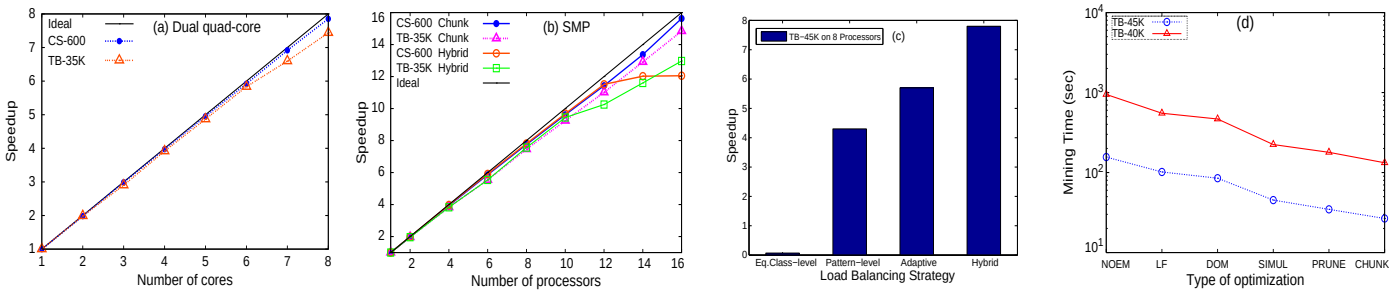

Figure 10: (a, b, c) Parallel performance; (d) Effect of optimizations

ble 1, other algorithms maintain relatively large 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 [9, 11].

7.7 Parallel Performance

The effectiveness of our parallel algorithms on a system with dual quad-core E5345 Xeon processors [10] is shown in Figure 10a. Our adaptive load balancing strategies achieved near-linear speedups up to 7.85-folds on Cslogs and 7.43-folds on Treebank, when all 8 cores of the system were used. Since the number of cores in future generation of multicores is likely to increase, we considered a 16-node SMP system [11] to test the scalability of our techniques. As shown in Figure 10b, the speedup continues to increase with the number of processors giving up to 15.5-fold speedup when using all the 16 available processors.

Load balance achieved by individual strategies for TB-45K is demonstrated in Figure 10c. Also, Figure 10b suggests that the need for more fine-grained strategies increases as one increases the number of processors. In case of CS-600, the performance of hybrid strategy (HyP) peaks at 12 processors and reaches its plateau ("CS-600 Hybrid" in Fig. 10b). This is due to a 6-node pattern that had approximately 33 million matches in a single database tree of 99 nodes, whose mining takes up to 45*sec*. As per Amdahl's law, hybrid strategy can never perform better than 45*sec* as it is limited by the task of finding extensions from a single tree. Thereafter the efficiency can only be improved by employing more fine-grained strategies such as the one in Section 6.3. Similarly, the speedup in case of TB-35K was saturated at 16 processors.

[10] 6GB RAM, 8MB shared L2, and 1333 MHz bus speed.

[11] 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

Cores (|C|)	1	2	4	8	16
N_t	1	1	3	4	3
N_d	2	3	6	5	5
N_c	0	0	0	0	7

Table 3: Cost analysis on CS-600, $\|FS\|$=252

Table 2 shows the average cost incurred in context switches taken over 10 different runs of a TB-35K experiment. For a given granularity g, $\sum_S N(g,S)$ is denoted as N_g in the table. With a single thread, the number of context switches is *zero* since the work is not shared at any level. As the number of cores are 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 the numbers in Table 2 are much lower than their theoretical upper bounds from Section 6.4: N_t=48 $\ll$ $|FS|$=451; N_d=11 $\ll$ 451;

Cores	1	2	4	6	8
EqP	1.00	1.61	1.94	1.95	2.01
AdP	1.00	1.77	2.23	2.25	2.30

Table 4: Speedups of parallel TreeMiner on TB-$45K$

and N_c=19 $\ll$ 451*(|C|-1), where |C| is number of cores.

Similar results on CS-600 can be found in Table 3. Even though the number of context switches is much smaller than that of in Table 2, it demonstrates the need for fine-grained strategies. A majority of run time (up to 90%) while mining Cslogs is spent in finding extensions from a very small subset of trees. Therefore, it is sufficient to parallelize the process of mining from this small subset of trees in order to realize good speedup. Without our fine-grained HyP and Chunk-level strategies, it is impossible to obtain the parallel efficiency that is shown in Figure 10a-b.

The results in Figure 10 are obtained by employing global job pools. We expect the contention overhead due to mutually exclusive accesses to job pools to be small as the locking on emerging architectures is likely to be very cheap [12]. If needed, one can seamlessly implement distributed or even hierarchical job pools with our scheduling service presented in Section 6.5.

We also parallelized TreeMiner by applying our task-level strategies EqP and AdP (see Table 4). Scope-lists in TreeMiner introduce dependencies which make it difficult to implement more fine-grained strategies. Since these lists are maintained in sorted order, data-partitioning methods (which construct the lists in parallel) likely to incur significant synchronization overheads. Furthermore, excessive use of dynamic data structures in TreeMiner significantly degrades the parallel performance as they impose a serial order on heap accesses – the system time (obtained from the "time" command) increased by more than 4 times when using all 8 cores. Here the techniques such as *memory pooling* are likely to be *ineffective* since these dynamic data structures often grow arbitrarily in size. These results re-emphasize the following key strategies for achieving 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.

8. DISCUSSION

With respect to memory-related optimizations, improving locality (spatial or temporal) continues to be important but in addition, bandwidth considerations must also be considered when designing data-intensive algorithms for emerging multicore systems. 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. Taken in isolation each optimization may not amount to a significant improvement on its own but the specific orchestration applied when combining them yield significant savings – L1 misses reduced by up to 1,442 times, memory footprints reduced by up to 366 times, bandwidth pressure reduced significantly while making making uniform small-sized accesses to main memory, and overall running time reduced by up to a factor of four on sequential execution.

It is also worth noting that the memory related optimizations have broader utility. The optimizations associated with the creation of on-the-fly embedding lists have found utility in problems such as XML indexing. Chunking may help for other mining tasks such as graph mining. All the optimizations discussed are also useful when mining induced subtrees – we get a 15-fold speedup when compared to FreqT (see Section 7.5). They can also potentially be leveraged for answering reachability queries in directed graphs, direction of research we are actively pursuing.

With regards to task scheduling, algorithms that can adapt and mold are essential to achieve performance commensurate with the number of cores in emerging multicore systems. Coarse-grained task partitioning is obviously preferred when the load is evenly balanced since the overheads associated with task management is minimal. However, often due to systemic, parametric or data-driven constraints, workload estimation is often challenging. In such scenarios the ability of an algorithm to adaptively modulate between coarse grained and finer grained task partitioning is essential to parallel efficiency. In fact how much an algorithm can adapt essentially dictates when the performance plateau is reached as we observe from our study. Our adaptive design strategy yielded linear speedup over our optimized sequential algorithm on a state-of-the-art CMP system (8 cores) and a modern SMP system (16 processors).

A key outcome here beyond the specific algorithm realized for tree mining is the realization of a general purpose scheduling service that supports the development adaptive and moldable algorithms for database and mining tasks. We are currently investigating ways to extend this service on modern cluster systems where each node is a CMP system.

Acknowledgments: This work is supported in part by NSF grants NGS-CNS-0406386, CAREER-IIS-0347662, RI-CNS-0403342, and CCF-0702587.

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

9. REFERENCES

[1] T. Asai *et al*. Efficient Substructure Discovery from Large Semi-structured Data. *In SDM*, pages 158–174.

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

[3] E. Charniak. Tree-bank grammars. *In AAAI*, 2:1031–1036, 1996.

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

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

[6] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *In FOCS*, pages 285–297, 1999.

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

[8] K. Hashimoto *et al*. A new efficient probabilistic model for mining labeled ordered trees. *In SIGKDD*, pages 177–186, 2006.

[9] 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.

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

[11] C. McNairy and R. Bhatia. Montecito: a dual-core, dual-thread Itanium processor. *In IEEE Micro*, 25(2):10–20, 2005.

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

[13] S. Nijssen and J. Kok. A quickstart in frequent structure mining can make a difference. *In SIGKDD*, pages 647–652, 2004.

[14] U. Ruckert and S. Kramer. Frequent Free Tree Discovery in Graph Data. *In ACM symposium on Applied computing*, pages 564–570, 2004.

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

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

[17] 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.

[18] S. Tatikonda and S. Parthasarathy. Mining Tree Structured Data on Multicores. Technical Report, The Ohio State University, 2007.

[19] S. Tatikonda, S. Parthasarathy, and M. Goyder. Lcs-trim: dynamic programming meets xml indexing and querying. In *In VLDB*, pages 63–74, 2007.

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

[21] 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.

[22] 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.

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

[24] C. Wang *et al*. Efficient Pattern-Growth Methods for Frequent Tree Pattern Mining. *In PAKDD*, pages 441–451, 2004.

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

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

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

[28] M. Zaki. Efficiently mining frequent trees in a forest. *In SIGKDD*, pages 71–80, 2002.

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

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