Parallel Formulations of Tree-Projection-Based Sequence Mining Algorithm *

Valerie Guralnik and George Karypis

Department of Computer Science and Engineering
University of Minnesota, Minneapolis, MN 55455
Technical Report: 03–0003

{guralnik, karypis}@cs.umn.edu

Abstract

Discovery of sequential patterns is becoming increasingly useful and essential in many scientific and commercial domains. Enormous sizes of available datasets and possibly large number of mined patterns demand efficient, scalable, and parallel algorithms. Even though a number of algorithms have been developed to efficiently parallelize frequent pattern discovery algorithms that are based on the candidate-generation-and-counting framework, the problem of parallelizing the more efficient projection-based algorithms has received relatively little attention and existing parallel formulations have been targeted only toward shared-memory architectures. The irregular and unstructured nature of the task-graph generated by these algorithms and the fact that these tasks operate on overlapping sub-databases makes it challenging to efficiently parallelize these algorithms on scalable distributed-memory parallel computing architectures. In this paper we present and study a variety of distributed-memory parallel algorithms for a tree-projection-based frequent sequence discovery algorithm that are able to minimize the various overheads associated with load imbalance, database overlap, and interprocessor communication. Our experimental evaluation on a 32 processor IBM SP show that these algorithms are capable of achieving good speedups, substantially reducing the amount of the required work to find sequential patterns in large databases.

Keywords: frequent sequential patterns, database projection algorithms, data mining, parallel processing

1 Introduction

In recent years there has been an increased interest in using data mining techniques to extract interesting patterns from sequential and temporal datasets. Sequence data arises naturally in many applications. For example, marketing and sales data collected over a period of time provide sequences that can be analyzed and used for projections and forecasting. Similarly, many important knowledge discovery tasks in genomics require the analysis of DNA and protein sequences to discover frequently occurring nucleotide or amino acid subsequences (commonly referred to as *motifs*) that correspond to evolutionary conserved functional units. The most time consuming operation in discovering such patterns is the computation of the occurrence frequency of all sub-sequences (called sequential patterns) in the sequence database. Unfortunately, the number of sequential patterns grows exponentially and for this reason various approaches have been developed [2, 14, 27, 10] that try to contain the complexity by imposing various temporal constraints and by considering only those candidates that satisfy a user specified minimum support constraint. However, even with these constraints, the task of finding all sequential patterns requires a large amount of computational resources (*i.e.*, time and memory), making it an ideal candidate for parallel processing.

The algorithms that discover sequential patterns are mainly motivated by those developed for frequent itemset discovery. In particular, there are three main classes of sequential pattern discovery algorithms. The first class of algorithms [27, 10], was developed by extending the *Apriori* [2] algorithm and its variants. These algorithms find the frequent patterns in a level-wise fashion using a candidate-pattern generation approach followed by an explicit frequency counting. The second class of algorithms decompose the lattice of frequent sequences into equivalence classes [32, 31] and compute the frequency of the patterns by using a vertical database format and set-intersection operations. The third class of algorithms [21, 24] was developed by extending database projection-based techniques [1, 9] that find the patterns by growing them one item/sequence at a time, and simultaneously partitioning (*i.e.*, projecting) the original database into pattern-specific sub-databases (which in general overlap). The process of pattern-growth and database-projection is repeated recursively until all frequent patterns are discovered. In general, projection-based frequent pattern discovery algorithms have been shown to substantially outperform those based on the level-wise paradigm; nevertheless, they still require a substantial amount of time.

A number of efficient and scalable parallel formulations have been developed for finding frequent itemsets and sequences that are based on the candidate-generation-and-counting framework [3, 18, 22, 16, 4], both for shared- and distributed-memory parallel computers [2, 22, 17, 8, 25, 29, 20]. However, the problem of parallelizing equivalence class-based and projection-based algorithms has received relatively little attention and existing parallel formulations for them have been targeted only toward shared-memory architectures [30, 28]. However, the irregular and unstructured nature of the task-graph that these algorithms generate and the fact that these tasks operate on overlapping sub-databases makes it challenging to develop efficient and scalable parallel formulations for distributed-memory parallel computers.

This paper attempts to improve our levels of understanding as to what are the best approaches by which to parallelize projection-based algorithms for pattern discovery on distributed-memory parallel computers. Toward this goal, we developed and studied a variety of algorithms for distributing the work among the processors so that the various overheads due to processor idling, database overlap, and interprocessor communication, are minimized. These algorithms try to achieve the desired goals by either distributing the work in a static or dynamic fashion. Specifically, we developed two static distribution algorithms that decompose the overall computation by exploiting either the data- or task-parallelism that is available in the problem. Our data-parallel formulation decomposes the computation by partitioning the database across the different processors, whereas the task-parallel formulation decomposes the computation by using task-decomposition to partition the lattice of frequent patterns. Finally, our dynamic distribution algorithm improves upon the task-decomposition-based algorithm by dynamically load-balancing the overall computation when the static task assignment leads to load imbalances. One of the key contributions of our research is the development of task decomposition schemes that use bipartite graph partitioning to simultaneously balance the computations and at the same time reduce the data sharing overheads by minimizing the portions of the database that needs to be shared by different processors. This scheme is used extensively in our static and dynamic load balancing algorithms.

We experimentally evaluated these algorithms using a projection-based algorithm for finding sequential patterns derived from the serial tree-projection algorithm for itemset discovery [1]. Our experiments on a 32-processor IBM

SP2 showed that these algorithms achieve good parallel performance and are effective in parallelizing the underlying computations. In particular, in the case of the data-parallel formulation our experimental results showed that it achieves good load-balance and leads to good speedups as the number of processors is increased. In the case of the static task-parallel formulation our experimental results showed that by using the bipartite graph-partitioning approach to statically divide the work among the processors we are able to dramatically reduce the overlap between the portions of the database that are needed by each processor while achieving reasonably good load-balance. Finally, in the case of the dynamic task-parallel formulation our experimental results showed that it can effectively load-balance the computations in cases in which static task assignment leads to load imbalances. Our parallel complexity analysis showed that due to the communication overhead incurred by the data-parallel formulation, its parallel performance is scalable only when the overall work is increased by increasing the size of the database and it does not scale well when the work increases due to a decrease of the minimum support value. On the other hand, our analysis showed that the task-parallel formulations do not suffer from this problem and scale well in both of these cases.

The rest of the paper is organized as follows. Section 2 provides some background definitions and introduces the notation that we will be using through-out the paper. Section 3 provides an overview on related serial and parallel work. Section 4 describes the serial tree-projection based algorithm for finding sequential patterns that we will use for our parallel formulation. Section 5 describes the various parallel algorithms that we developed for parallelizing the tree-projection algorithm. Section 6 analyzes the parallel runtime and scalability of these algorithms. Section 7 presents the experimental evaluation of the different algorithms. Finally, Section 8 provides some concluding remarks.

2   Sequence Mining: Definitions and Notation

We will use the itemset model [3] and sequence model [27], both of which were introduced by Agrawal and Srikant. These two models are defined as follows. Let $I = \{i_1, i_2, \ldots, i_n\}$ be the set of all items. An ***itemset*** is a subset of I. A ***sequence*** $s = \langle t_1, t_2, \ldots, t_l \rangle$ is an ordered list of itemsets, where $t_j \subseteq I$ for $1 \leq j \leq l$. A sequential database $\mathcal{D}$ is a set of sequences. The length of a sequence s is defined to be the number of items in s and is denoted by $|s|$. Similarly, the length of an itemset t is defined to be the number of items in t and is denoted by $|t|$. Given a sequential database $\mathcal{D}$, $|\mathcal{D}|$ denotes the number of sequences in D. We assume that the items in I can be arranged in a lexicographic order, and we will use consecutive integers starting from one to represent the items according to that ordering.

Sequence $s = \langle t_1, t_2, \ldots, t_l \rangle$ is called a ***sub-sequence*** of sequence $s' = \langle t_1', t_2', \ldots, t_m' \rangle$ $(l \leq m)$ if there exist l integers $i_1, i_2, \ldots i_l$ such that $1 \leq i_1 < i_2 < \ldots < i_l \leq m$ and $t_j \subseteq t_{i_j}'$ $(j = 1, 2, \ldots, l)$. If s is a sub-sequence of s', then we write $s \subseteq s'$ and say sequence s' ***supports*** s. Given a sequence s let $\mathcal{D}_s \subseteq \mathcal{D}$ be the set of database sequences that support s. The ***support*** of a sequence s is defined to be $|D_s|/|D|$ and denoted by $\sigma_{\mathcal{D}}(s)$. From the definition, it always holds that $0 \leq \sigma_{\mathcal{D}}(s) \leq 1$ for any sequence $s \in \mathcal{D}$.

We will use the traditional method for writing out sequences, in which each itemset is represented as a list of items ordered according to their lexicographical order and enclosed within matched parentheses (), and the sequence of these itemsets is written one-after-the-other enclosed within matched angled parentheses $\langle \rangle$.

To illustrate the above definitions consider the set of items $I = \{1, 2, 3\}$. This set can generate seven possible itemsets and each of them is represented as (1),(2),(3),(1, 2), (1, 3),(2, 3),(1, 2, 3). Let $t_1 = (1, 2)$, $t_2 = (1, 2, 3)$, and $t_3 = (3)$, be three itemsets of sizes two, three, and one, respectively. Sequence $s = \langle t_1, t_2, t_3 \rangle = \langle (1, 2), (1, 2, 3), (3) \rangle$ has three itemsets and has length $|s| = 2 + 3 + 1 = 6$. Sequence $s' = \langle (1, 3), (1, 2, 3), (1, 2, 3), (2), (2, 3) \rangle$ supports s, or in other words s is a sub-sequence of s'.

The problem of finding frequent sequential patterns given a minimum support constraint [27] is formally defined as follows:

> **Definition 1 (Sequential Pattern Mining)** *Given a sequential database $\mathcal{D}$ and a user-specified parameter σ ($0 \leq \sigma \leq 1$), find all sequences each of which is supported by at least $\lceil \sigma |D| \rceil$ sequences in $\mathcal{D}$.*
> ∎

The user-specified parameter σ is called the ***minimum support constraint***, and the discovered sub-sequences are called ***frequent sequential patterns***.

3 Related Research

Efficient algorithms for finding frequent itemsets or sequences in very large transaction or sequence databases have been one of the key success stories of data mining research. These frequently occurring patterns can be used to find association rules and/or to extract prevalent patterns that exist in the datasets. One of the early computationally efficient algorithm was Apriori [3], which finds frequent itemsets of length l based on previously generated $(l - 1)$-length frequent itemsets. The GSP [27] algorithm extended the Apriori-like level-wise mining method to find frequent patterns in sequential datasets. The basic level-wise algorithm has been extended in a number of different ways leading to more efficient algorithms such as DHP [19, 18], Partition [22], SEAR and Spear [16], and DIC [4]. An entirely different approach for finding frequent itemsets and sequences are the equivalence class-based algorithms Eclat [32] and SPADE [31] that break the large search space of frequent patterns into small and independent chunks and use a vertical database format that allows them to determine the frequency by computing set intersections. More recently, a set of database projection-based methods has been developed that significantly reduces the complexity of finding frequent patterns [1, 9, 21, 23, 24]. The key idea behind these approaches is to find the patterns by growing them one item at a time, and simultaneously partitioning (*i.e.*, projecting) the original database into pattern-specific sub-databases (which in general overlap). The process of pattern-growth and database-projection is repeated recursively until all frequent patterns are discovered. Prototypical examples of such algorithms are the tree-projection algorithm [1] that constructs a lexicographical tree and projects a large database into a set of reduced, item-based sub-databases based on the frequent patterns mined so far. Another similar algorithm is the FP-growth algorithm [9] that combines projection with the use of the FP-tree data structure to compactly store in memory the transactions of the original database. The basic ideas in this algorithm were recently used to develop a similar algorithm for finding sequential patterns [21].

A number of parallel frequent itemset discovery algorithms have been developed that focus on parallelizing the various serial algorithms for that problem [2, 22, 17, 8, 25, 29, 20, 28]. Depending on the nature of the underlying serial algorithm, many of these approaches follow the same parallelization strategy. Providing exact details on all of these algorithms is beyond the scope of this section, and for this reason we only focus on the various issues involved in parallelizing serial algorithms that involve candidate generation followed by an explicit frequency counting (*e.g.*, Apriori, DHP, Partition, SEAR, DIC), as they are illustrative of the various issues related to parallel processing that need to be considered.

There are three general approaches for parallelizing this type of computations. One is to replicate the candidates on all processors [2, 22, 17], the other is to avoid replication [2, 8, 25], and the third to allow partial replication [8, 25]. The first approach is commonly referred to as *Count Distribution* (CD) approach [2, 8]. The CD approach parallelizes frequent itemset discovery algorithms by replicating the candidate generation process on all the processors and parallelizing the counting process. At each iteration after each processor completed a counting phase, the global counts of candidates are computed by summing individual counts using a global reduction operation. However, as the number of processors increases, the CD approach suffers from synchronization and communication overhead. Furthermore, if the number of candidate itemsets is large they may not fit into the main memory. The second approach, commonly referred to as Data Distribution (DD) [2], attempts to address the problems associated with CD by simultaneously partitioning the candidates and the database among the processors. The DD approach exploits the total available memory better than CD, as it partitions the candidate set among processors. As the number of processors increases, the number of candidates that the algorithm can handle also increases. However, the performance of the approach suffers from the need of each processor to communicate the locally stored transactions to every processor in the system. This can be substantially decreased by performing the data distribution in an intelligent fashion so that to reduce the sharing of the databases, leading to the IDD scheme [8]. The third approach, called *Hybrid* tries to simultaneously solve the problems associated with CD and IDD by partially replicating the candidates [8]. This scheme combines the best elements of CD and IDD and has been shown analytically and experimentally to outperform either one of them.

While parallel itemset discovery algorithms has attracted wide attention, there has been relatively little work on parallel mining of sequential patterns. Shitani *et al.* [26] introduced three parallel versions of the GSP algorithm called NPSPM, SPSPM and HPSPM that were designed for distributed-memory parallel computers. All versions partition the dataset into equal blocks among the processors. NPSPM is based on the CD approach and suffers from the same

drawbacks as CD. SPSPM and HPSPM are based DD method, with HPSPM using a more intelligent strategy to partition candidate sequences. All three schemes involve exchanging the remote database partitions during each iteration, resulting in high communication cost and synchronization overhead. Zaki [30] presented various parallel formulations of the SPADE algorithm that were designed for shared-memory parallel computers. The SPADE algorithm decomposes the search space of sequential patterns into independent classes, and thus is well-suited for parallel processing. He presented both data- and task-parallel formulations of SPADE and showed that the latter performed better. One of the key contributions of this work is the use of a dynamic load-balancing scheme to ensure that each processor gets an equal amount of work. The dynamic load-balancing followed a *work-pool* model which resulted in good performance as the underlying parallel architecture was shared-memory and the database was shared among the processors.

4   Serial Tree-Projection Algorithm

The parallel algorithms in this paper are based on the tree projection algorithm for frequent itemset discovery developed by Agarwall *et al* [1]. This algorithm uses a lexicographic tree, called the ***projection tree***, to represent the entire set of itemsets that can be constructed from a set of items I. Each node in this tree corresponds to a particular itemset such that the level of the node in the tree corresponds to the length of the itemset. That is, nodes at level k correspond to itemsets of length k. Given a particular node u at level k and its associated itemset $(u_{i_1}, u_{i_2}, \ldots, u_{i_k})$ in which the various items u_{i_j} are shown in lexicographically increasing order, then the children of u in the tree correspond to all itemsets $(u_{i_1}, u_{i_2}, \ldots, u_{i_k}, u_{i_{k+1}})$ such that item $u_{i_{k+1}}$ is lexicographically larger than u_{i_k}. The children represent itemsets of length $k + 1$ and are called their parent's extensions. In addition, the children of a node are ordered lexicographically in terms of their itemsets. For example, if $I = \{1, 2, 3, 4\}$, the itemset (1) has three children: $(1, 2)$, $(1, 3)$, and $(1, 4)$, whereas the itemset (3) has only one child $(3, 4)$. From the above description it can be easily seen that there is a one-to-one mapping between tree nodes and itemsets (*i.e.*, the projection tree represents all possible itemsets and each itemset corresponds to only a single node of the tree).

The tree-projection algorithm grows the tree such that only the nodes corresponding to frequent itemsets are generated. In principle, the tree can be grown in either breadth-first or depth-first fashion, even though the original algorithm focused mostly on the breadth-first approach [1]. The breadth-first version of the algorithm consists of a number of iterations in which it grows the tree level-by-level. In particular, during the kth iteration, it extends all the frequent itemsets corresponding to the nodes at level $(k - 1)$ in order to obtain the *candidate* itemsets of length $(k + 1)$ corresponding to the nodes of level $(k + 1)$. Thus, it performs the extensions by growing the length of the itemsets by two. This is done for computational efficiency purposes as it leads to better cache utilization [1]. In principle, the candidate extensions of a given node can be derived from the items that are lexicographically greater than the largest item of that node. However, because of the downward closure property of the minimum support constraint, this set can be substantially reduced by using only the items that were part of frequent extensions of its parent node [1].

All the nodes that belong to a sub-tree which potentially can be extended are called ***active extensions***. If a node cannot be extended any further it becomes *inactive* and is pruned from the tree. Figure 1 illustrates an example of projection tree that has just been expanded to level 3. In this example the only active nodes are $()$, (1), $(1, 2)$ and $(1, 2, 4)$. The other nodes could not be extended any further and therefore became inactive.

One of the key features of the tree-projection algorithm is that the support of the candidate itemsets that are being generated during the kth iteration of the algorithm, is determined by using a *database projection* approach that *projects* the original transactions at the parent nodes of level $k - 1$. This is done as follows. For each node of the tree, the algorithm maintains a list of items that can be found in the itemsets corresponding to its descendant nodes. These items are called ***active items*** of that node. When a transaction is projected to a node, only the items that occur in its active item list are kept, and the remaining are eliminated. This process is done efficiently by performing the projection in a recursive fashion along a path from the root to that node that goes through active extensions. The key idea behind this approach is to percolate down the tree only those items in a transaction that can potentially be useful in extending the tree by one more level. With every pass of the algorithm, many extensions become progressively inactive, which in turn results in reducing the number of active items associated with the nodes. Thus, the size of the projected transaction set progressively reduces. This yields the algorithm its efficiency in the counting phase.

The actual process of determining the frequency of candidate itemsets of length $(k + 1)$ is accomplished in the
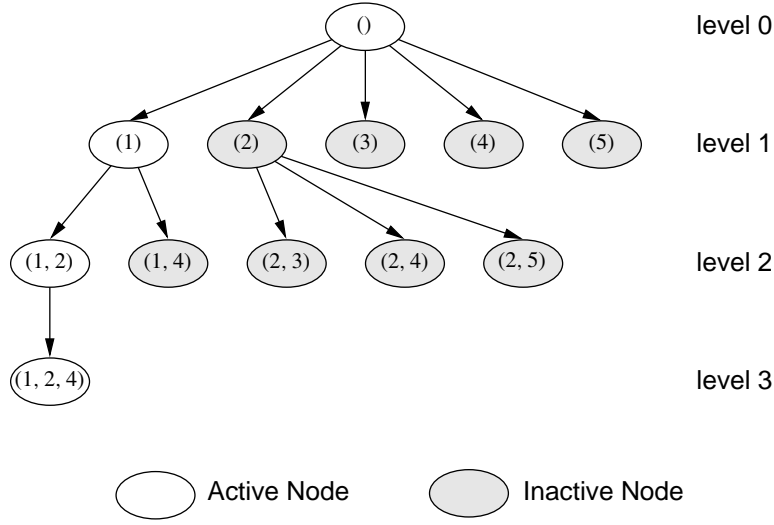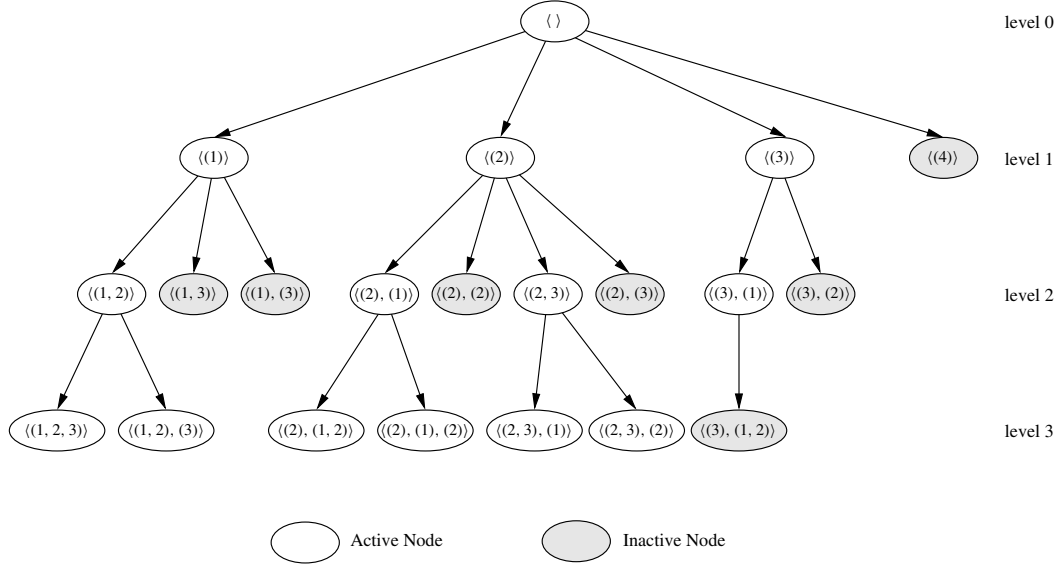
Figure 1: An example of a projection tree for itemsets.

following way. Each node at level $k - 1$, corresponding to the itemset $u = (u_{i_1}, u_{i_2}, \ldots, u_{i_{k-1}})$, maintains a *count matrix*, which is used to count the frequency of candidate extensions $(u_{i_1}, u_{i_2}, \ldots, u_{i_{k-1}}, u_{i_k}, u_{i_{k+1}})$, where items u_{i_k} and $u_{i_{k+1}}$ are active extensions of that node. Once the transactions have been projected to the node, the algorithm iterates through them to determine the frequency of the candidate patterns.

4.1   Tree-Projection Algorithm for Sequential Pattern Discovery

The tree-projection algorithm for finding sequential patterns also uses a lexicographic tree to represent all possible sequences. Each node of that tree corresponds to a particular sequence such that nodes at level k correspond to sequences of length k. The relationship between the sequences of a node and that of its children is similar in nature to the relationship between the itemsets in the original tree-projection algorithm. However, the fact that a sequence can grow by either adding an extra item in its last itemset or by adding a new one-item itemset, adds an additional level of complexity on the type of relations that exist. In particular, each node has two types of children. The first type corresponds to nodes whose sequences were obtained by adding an item in the last itemset of that sequence. This item has to be lexicographically larger than all the items in that last itemset. The second type corresponds to nodes whose sequences were obtained by adding a new one-item itemset at the end of the sequence. There is no lexicographic-related restrictions on the item of this one-item itemset. The first type of nodes are said to be obtained by performing an ***itemset extension***, whereas the second type of nodes are said to be obtained by performing a ***sequence extension***. For example, if $I = \{1, 2, 3, 4\}$, then the node with sequence $\langle (2) \rangle$ has as children the nodes with sequences $\langle (2, 3) \rangle$ and $\langle (2, 4) \rangle$ obtained via itemset extension and the nodes with sequence $\langle (2)(1) \rangle$, $\langle (2)(2) \rangle$, $\langle (2)(3) \rangle$, and $\langle (2)(4) \rangle$ obtained via sequence extension.

This tree is grown in a fashion similar to that discussed in Section 4 such that only the nodes corresponding to frequent sequences are generated. Moreover, a similar bi-level candidate sequence generation approach is used, in which the nodes at level $k - 1$ are responsible for generating the candidate sequences at level $k + 1$. The frequency of these candidate sequences is determined by using a projection approach in which the database sequences are progressively being projected along the path from the root to the node at level $k - 1$. However, since the sequence of a node can be extended by either itemset or sequence extension, there are two sets of active items that are maintained; the one that can be used for itemset extensions and the other to be used for sequence extensions. During projection, any item that is not part of either of these two sets is eliminated. Figure 2 illustrates an example of the projection tree for sequence mining. In this example the set of active items of node $\langle (2) \rangle$ is $\{1, 3\}$, where $\{1\}$ is an active sequence extension and $\{3\}$ is an active itemset extension.

6

Figure 2: An example of the projection tree for sequential patterns.

The frequency of the various sequences of length $k + 1$ is determined as follows. Each node, representing the sequence $s = \langle t_1, t_2, \ldots, t_m \rangle$, at level $k - 1$ maintains four count matrices. The first matrix is a lower-triangular matrix, which is used to count the frequency of candidate extensions $\langle t_1, t_2, \ldots, (t_m, i, j) \rangle$, where items i and j are active itemset extensions of the node. The second matrix is used to count the frequency of candidate extensions $\langle t_1, t_2, \ldots, t_m, (i), (j) \rangle$, where items i and j are active sequence extensions of the node. The third matrix is a lower-triangular matrix, which is used to count the frequency of candidate extensions $\langle t_1, t_2, \ldots, t_m, (i, j) \rangle$, where items i and j are active sequence extensions of the node such that item j is lexicographically larger then item i. The last matrix is used to count the frequency of candidate extensions $\langle t_1, t_2, \ldots, (t_m, i), (j) \rangle$, where item i is an active itemset extension of the node and item j is an active sequence extension of the node.

Once each sequence has been projected to a particular node at level $k - 1$, the algorithm updates the frequencies of the corresponding matrices. These updates can be done efficiently by using a sparse-matrix representation of the sequence in which each row corresponds to an itemset and each column corresponds to a distinct item present in the sequence.

5 Parallel Formulations of the Tree Projection Algorithm

The overall structure of the computations performed by the serial tree-projection algorithm is encapsulated in the tree that it generates. In particular, using the breadth-first expansion of the tree, the computation proceeds as follows. Initially, there is a single node (*i.e.*, the NULL node), which is expanded to generate all sequential patterns of length one – leading to a tree of depth one. Then, the nodes of that tree that correspond to sequential patterns whose support satisfy the minimum support constraint are expanded to generate sequential patterns of length two – leading to a tree of depth two. This process of level-by-level tree expansion continues until the tree cannot be grown any further because none of the leaf nodes (*i.e.*, candidate sequential patterns) satisfy the minimum support constraint. The bulk of the computation in this algorithm is performed while determining which of the sequences (*i.e.*, nodes in the tree) satisfy the minimum support constraint. As discussed in Section 4.1, this is achieved by projecting the original sequences into each one of the nodes of the tree and then counting the frequencies using the various matrices.

Given that the computations are structured in this fashion, there are two general ways that can be used to decompose them [7]. The first approach exploits the data parallelism that exists in computing the various frequencies at each node at level $k - 1$, whereas the second approach exploits the task parallelism that exists as a result of the tree-based nature

7

of the overall computation. In the rest of this section we present a number of different parallel formulations that exploit these sources of parallelism.

In developing these parallel formulations we assume that the underlying architecture is that of a distributed-memory parallel computer with explicit message-passing the only means of interprocessor communication. Moreover, we assume that the database is too large to fit in memory and thus resides on the disk. We assume that each processor has access to a local disk and that any access to data stored in other processor's disks requires an explicit point-to-point communication operation between the processor who *owns* the disk and the processor who needs to access the data.

5.1 Data Parallel Formulation

The idea behind the data parallel formulation (DPF) is to divide between the different processors the computations required to determine the frequency of the various sequences at each node of the tree. To a large extent, this parallel formulation is similar in nature to the count distribution method developed for parallelizing the serial Apriori algorithm for finding frequent itemsets [2, 8] described in Section 3.

Our parallel formulation works as follows. If p is the total number of processors, the original database is initially partitioned into p equal-size parts, each one is assigned to a different processor, and stored in its local disk. The processors then proceed to cooperatively grow the tree in a breadth-first fashion one-level-at-a-time. The frequencies of the candidate sequences corresponding to the nodes of the tree at level $k + 1$ are computed in two steps. During the first step, each processor determines the frequencies using only its local subset of the database by projecting each one of them to the nodes at level $k - 1$. During the second step, the overall frequencies are determined by using a reduction operation to add up the individual frequencies. These global frequencies are made known to all the processors and are used to determine which nodes at the $(k + 1)$st level meet the minimum support constraint. Note that in this approach, all the processors build the same tree, which is identical to that built by the serial algorithm.

In general, since the database is equally distributed among the processors, the overall computation will be well-balanced. However, there may be some pathological cases in which a block-partitioning of the database can lead to load imbalances due to the fact that certain processors are assigned sub-databases that have more frequent patterns than others. This can be avoided by using either a block-cyclic or a random partitioning [7] that, on the average, eliminate such pathological cases.

5.2 Task Parallel Formulation

The tree-projection algorithm computes the frequency of the candidate patterns at level $k+1$ by projecting the database sequences at the various nodes of the $k - 1$ level. Once the database has been projected to each one of these nodes, the actual frequency counting can proceed at each node independently. Thus, the computations at each node at level $k - 1$ becomes an independent task and the overall algorithm can be parallelized by distributing these tasks among the available processors.

The tasks associated with the various nodes of the tree can be distributed using either a *horizontal* or *vertical* scheme. In the horizontal scheme, each level of the tree is grown in a lock-step fashion and the various tasks (*i.e.*, nodes of the tree) are distributed among the processors using a different decomposition for each level. Each of these decompositions is performed so that the work assigned to each processor is balanced. On the other hand, in the vertical scheme, each processor is assigned a set of tasks corresponding to certain nodes at a particular level $k + 1$ of the tree, and it then proceeds to independently grow the entire subtrees rooted under these nodes.

In general, the vertical scheme is better-suited for distributed-memory architectures whereas the horizontal scheme will lead to good performance only on shared-memory architectures. This is because of the following. Consider two nodes u and v located at two consecutive levels of the tree and let $\mathcal{D}_u$ and $\mathcal{D}_v$ be the set of database sequences that are projected onto these nodes. If there is a parent-child relationship between u and v, as it will be in the case of the vertical approach, then $\mathcal{D}_v \subseteq \mathcal{D}_u$. As a result, once the processor responsible for node u has finished working on that node, it already stores locally the database sequences that it needs in order to process v (as well as all nodes that are descendants of u). On the other hand, if there is no parent-child relationship between u and v, as it will in general be in the case of the horizontal approach, then $\mathcal{D}_u$ can be entirely disjoint from $\mathcal{D}_v$. Consequently, the processor may either spent additional communication time retrieving the sequences in $\mathcal{D}_v$ or being forced to replicate locally the entire

database $\mathcal{D}$. Because of this reason, our parallel formulations that exploit task parallelism follow the vertical scheme.

In the following two sections we present two different formulations that are based on the vertical scheme. The first distributes the nodes of the tree using a static assignment of tasks to processors that does not change during the execution of the algorithm, whereas the second allows for tasks to be dynamically moved between the processors if such moves improve the overall load balance.

5.2.1 Static Decomposition

Our static task-parallel formulation (STPF) distributes the tasks among the processors in the following way. First, the tree is expanded using the data-parallel algorithm described in Section 5.1, up to a certain level $k+1$, with $k > 0$. Then, the different nodes at level k are distributed among the processors. Once this initial distribution is done, each processor proceeds to generate the subtrees (*i.e.*, sub-forest) that are rooted at the various nodes that it has been assigned. Note that the algorithm distributes the nodes at level k (and not those of level $k + 1$) because during the frequency counting phase of the candidates at level $k + 2$ the database needs to be projected to the nodes at level k.

In order for each processor to proceed independently of the rest it must have access to the sequences in the database that may support the patterns corresponding to the nodes of the tree that it has been assigned. The database sequences (or portions of) that each processor P_i needs to have access to can be determined as follows. Let S_i be the set of nodes assigned to P_i, A_i be the union of all the active items of the nodes in S_i, and B_i be the union of all the items in each of the frequent patterns that correspond to a node in S_i. Given these definitions, then in order for each processor to be able to proceed independently of the rest it needs to have access to all the sequences that contain items belonging in the set $C_i = A_i \bigcup B_i$. Moreover, since it computes only the frequent patterns that correspond to nodes that are descendants of the nodes in S_i, it needs to only retain the items from the original sequences that are in C_i. We will refer to the set of items C_i as the ***sub-forest itemset*** of the node set S_i.

In our algorithm, once the distribution of the nodes at level k is determined, the sets $C_0, C_1, \ldots, C_{p-1}$ are determined and broadcast to all the processors. Each processor then reads the local portion of the database (the one that was used by the data-parallel algorithm), splits it into p parts, one for each processor, and sends it to the appropriate processor. Each processor, upon receiving the sequences corresponding to its portion of the tree, writes them to the disk and proceeds to expand its nodes independently. Note that since the sets C_i for different processors can overlap, processors will end up having overlapping sections of the original database.

Bin-Packing-Based Task Distribution The key step in the STPF algorithm is the method used to partition the nodes of the kth level of the tree into p disjoint sets $S_0, S_1, \ldots, S_{p-1}$. In order to ensure load-balance, this partitioning must be done in a way so that the work is equally divided among the different processors. A simple way of achieving this is to assign a weight to each node based on the estimated amount of work required to expand that node, and then use a bin-packing algorithm [5] to partition the nodes into p equal-weight buckets. This weight can be either a measure of the actual computational time that is required, or it can be a measure that represents a relative time, in relation to the time required by other nodes. Obtaining relative estimates is much easier than obtaining estimates of the actual execution time. Nevertheless, accurately estimating the relative amount of work associated with each node is critical for the overall success of this load-balancing scheme.

A simple estimate of the relative amount of work of a particular node is to use the support of its corresponding sequential pattern. The motivation behind this approach is that if a node has a high support it will most likely generate a deeper subtree than a node with a lower support. However, this estimate is based on a single measurement and can potentially by very inaccurate. A better estimate of the relative amount of work can be obtained by summing up the frequency of all of its children nodes at the $(k + 1)$st level of the tree that satisfy the minimum support constraint. This measurement, by looking ahead at the support of the patterns of length $k + 1$, will tend to provide more accurate estimates. This is the method that we use to estimate the relative amount of work associated with each node. Note that since our algorithm switches from data to task parallel when the tree has been expanded to level $k + 1$, the frequency of these nodes has already been computed.

Bipartite Graph Partitioning-Based Task Distribution Even though the bin-packing-based approach is able to load-balance the computations, it may lead to partitions in which the sub-forest itemsets assigned to each processor

have a high degree of overlap. Consequently, the follow-up database partitioning will also lead to highly overlapping local databases. This increases the amount of time required to perform the partitioning, the amount of disk-storage required at each processor, and as we will see in the experiments presented in Section 7.1, it also increases the amount of time required to perform the projection. Ideally, we will like to partition the nodes at the kth level in such a way so that in addition to balancing the load we also minimize the degree of overlap among the different databases assigned to each processor.

Since the degree of overlap in the local databases is directly proportional to the degree of overlap in the sub-forest itemsets, we can minimize the database overlap by minimizing the overlap in the sub-forest itemsets. This latter problem can be solved by using a minimum-cut bipartite graph partitioning algorithm, as follows.

Let $G = (V_A, V_B, E)$ be an undirected bipartite graph, where V_A, and V_B are the two sets of vertices and E is the set of edges. The vertices in V_A correspond to the nodes of the tree at level k, the vertices in V_B correspond to the sequence items, and there is an edge $(u, v) \in E$ with $u \in V_A$ and $v \in V_B$, if the item v is an active item of node u. Each vertex $u \in V_A$ has a weight $w(u)$ that is equal to the relative amount of work required to expand its corresponding subtree, and each vertex $v \in V_B$ has a weight of one. Furthermore, each edge (u, v) has a weight of one.

A partitioning of this bipartite graph into p parts that minimizes the edge-cut (*i.e.*, the number of edges that straddle partitions), subject to the constraint that the the sum of the weight of the vertices in V_A assigned to each partition is roughly the same, can be used to achieve the desired partitioning of the tasks. This is because since each partition contains tree-nodes whose total estimated work is roughly the same, the overall computation will be balanced. Furthermore, by minimizing the edge-cut the resulting partitioning groups nodes together so that their sub-forest itemsets have as little overlap as possible. Note that an edge belonging to the cut-set indicates that the corresponding item is shared between at least two partitions. In general, for each item u, the number of its incident edges that belong to the cut-set plus one represent the total number of sub-forest itemsets that this node belongs to.
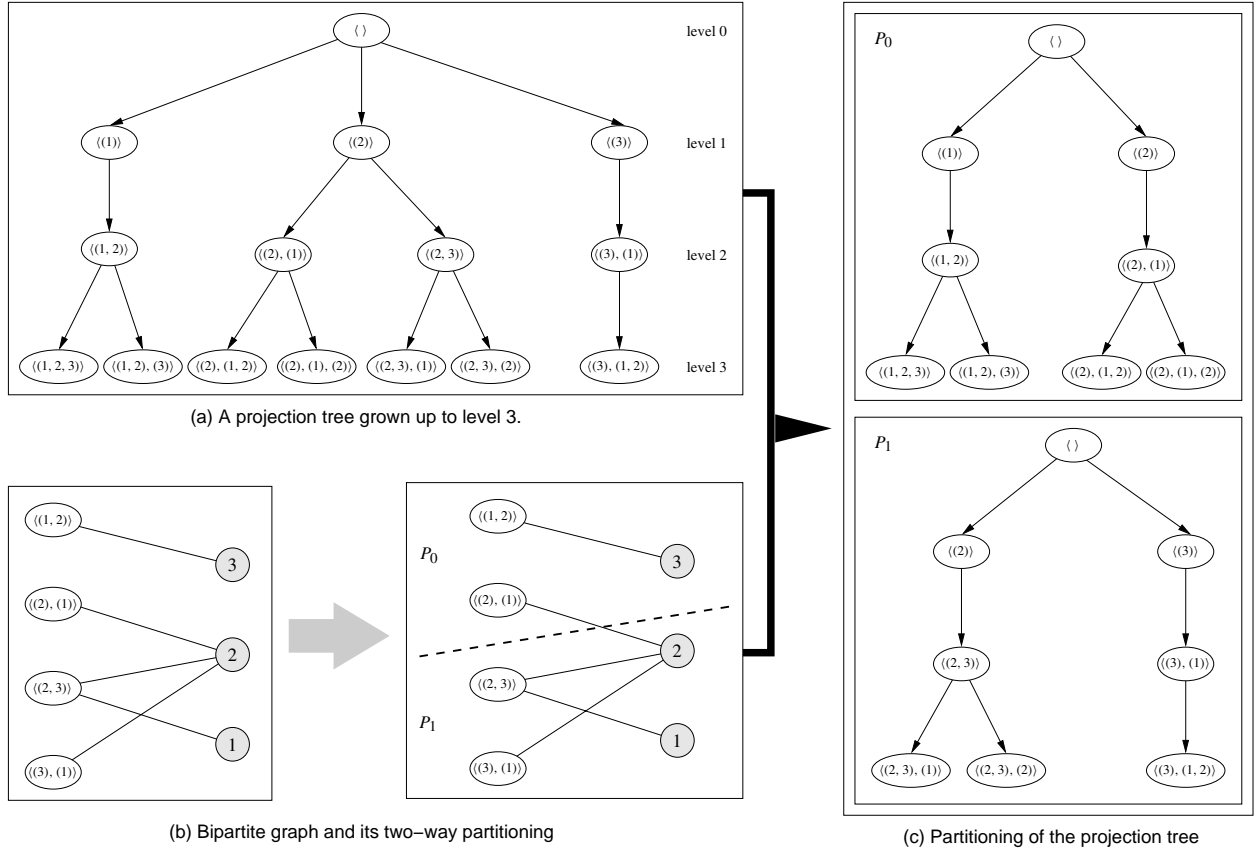
Figure 3 illustrates the graph-partitioning based using a small example. In particular consider the projection tree $\mathcal{T}$ shown in Figure 3(a), which has been expanded up to level three, and assume that $\mathcal{T}$ needs to be distributed to two processors using the nodes of the second level. The algorithm will then proceed to form the bipartite graph on the left of Figure 3(b) whose vertices are derived from the second level nodes and the active items of these nodes. Using this graph and assuming that the relative amount of work for each node is the same, a two-way min-cut partitioning will be computed that assigns nodes $\langle (1, 2) \rangle$ and $\langle (2), (1) \rangle$ to processor P_0 and nodes $\langle (2, 3) \rangle$ and $\langle (3), (1) \rangle$ to processor P_1. Finally, as illustrated in Figure 3(c), using this partitioning, each processor retains the portion of the tree that corresponds to the nodes assigned to it.

In our algorithm, we compute the bipartite min-cut partitioning algorithm using the multi-constraint graph partitioning algorithm [12] available in the METIS graph partitioning package [11].

5.2.2 Dynamic Load-Balancing

The underlying assumption for the efficient execution of STPF algorithm described in Section 5.2.1 is that the relative amount of work required to expand each one of the tree nodes can be accurately estimated beforehand. The experiments, presented in Section 7.2, show that even though estimates are in general accurate, their accuracy tends to decrease as the number of processors increases. Moreover, there exist data sets for which the estimates can be substantially wrong, even for small number of processors. For this reason we developed a dynamic task-parallel formulation (DTPF) that monitors the load-imbalance and moves work between the processors, as needed.

The topic of dynamic load-balancing of tree-based computations has been extensively studied in the context of parallel heuristic state-space search algorithms [7] and a number of different approaches have been developed. A simple way to load-balance the computation is do that in a synchronous fashion [7, 13]. In this approach after the nodes and database have been partitioned between processors, the processors synchronously extend the tree level-by-level. After each level expansion the processors communicate between each other to determine whether the work needs to be re-balanced. If the conclusion is made that workload re-distribution is necessary, then the processor P_i with lth maximum estimated workload W_i sends work to the processor P_j with lth minimum estimated workload W_j. The portion of the work to be send to the receiving processor can be determined randomly or the minimum cut bipartite graph partitioning approach described in Section 5.2.1 can be used to select the parts of the tree that lead to the least amount of data sharing overhead. This approach is similar in spirit to the horizontal task decomposition

(a) A projection tree grown up to level 3.

(b) Bipartite graph and its two−way partitioning

(c) Partitioning of the projection tree

Figure 3: An example of the bipartite graph-partitioning based approach for task decomposition on two processors. (a) A projection tree grown up to level three. (b) The bipartite graph formed from the second level nodes of the tree and the active items and a min-cut two-way partitioning of this graph. (c) The decomposition of the projection tree that is induced by the graph partitioning. partitioned bipartite graph

scheme described in Section 5.2.

Unfortunately, even though this approach is quite simple to implement, it does not significantly reduce the load-imbalance overhead. This is primary due to the fact that the number of possible load-balancing points is quite small (equal to the depth of the tree). To see this, consider a scenario in which a computation at certain level is particularly unbalanced. The imbalance will only be determined after the counting of candidate extensions has been completed at that level. If this computational step contributes to the majority of the parallel imbalance, the synchronous dynamic load-balancing scheme cannot do anything to correct it. Our experiments with this approach (not presented here) verified this observation, as we were not able to substantially reduce the load-imbalance overhead.

For this reason we developed dynamic load-balancing algorithms that use the asynchronous paradigm and are similar in nature to the task-parallel formulation algorithm described in Section 5.2.1. That is, the tree is extended to level $k+1$ using the data-parallel formulation and then the bipartite graph partitioning algorithm is used to partition the nodes at level k among the p processors. However, instead of allowing each processor to expand its different subtrees independently to the very end, the processors check to determine whether the work needs to be re-balanced.

To accomplish this we use a receiver initiated load-balancing with random pooling scheme [7]. In this scheme, as a processor becomes idle (that is it finishes its portion of allocated work), it randomly selects a donor processor and sends it a work request. The donor processor sends a response indicating whether or not it has additional work. If a response indicates that a donor does not have anymore work, the processor selects another donor and sends a work request to that donor. Otherwise, the processor receives nodes to expand along with the portion of the database associated with

those nodes. Upon receiving new work the processor starts extending newly received nodes. This process continues until every processor completely extended the nodes assigned to it. Dijkstra's token termination algorithm [6] is used to detect whether all processors have become idle and thus the overall computations have finished.

The key issue in this approach is when will the processors service work requests. The natural point at which an active processor can service a request is the time between the point it completed counting support of candidate extensions at a certain level and is ready to count support of candidate extensions at the next level. This is because, this is the time at which work can be transferred by sending only some of the nodes of the tree along with relevant portions of the locally stored databases. On the other hand, if we allow work transfered to take place while a processor is in the middle of counting the support of the candidate patterns at the next level, then besides the tree nodes and their respective portions of the database, we also need to send the partial counting matrices associated with the different nodes. This may substantially increase the amount of data that needs to be transferred, severely reducing the gains achieved by dynamic load-balancing.

However, the problem associated with servicing work requests only at the natural breaking points of the computation is the following. The bulk of the computation is performed during the counting phases, as a result if processor P_i sends a work request to processor P_j right after P_j has started its counting phase, it will take a significant amount of time before P_j services that request. Through out that time processor P_i will remain blocked, waiting for work.

Reducing Idling Overheads   One way of eliminating this idling is to follow a protocol in which processor P_j periodically checks for work requests (while working on its counting phase), and responds to them by indicating that even though it has work it cannot send it right away. Upon receiving such a response, processor P_i can either decide to wait for P_j or ask another processor for work.

This new load-balancing protocol can eliminate most of the waiting time incurred in receiving work. However, as discussed earlier, because the depth of the trees are quite small, the overall number of natural work transfer points is quite small. As a result, a processor can still be blocked for a long time. To address this problem we developed the following algorithm. Instead of servicing work requests only during the natural points, we follow a policy in which if a request comes during the early stages of each counting phase, then the processor can discard the work it has done so far and service the work request. Because the amount of work done so far is not substantial, this will not affect the overall time significantly. Instead the computation will benefit from improved load balance.

The overall structure of the developed algorithm is the following. If a work request comes at the early stages of a counting phase, an active processor discards the work done so far and shares its workload with the recipient. If a work request comes at the middle stages of the counting phase, an active processor lets the receiver know that while it has work, the work is not available immediately. Along with this message, the active processor also sends an estimated time before completion as well as an estimated relative amount of workload at the next level. Toward the end of the counting phase, an active processor ignores the work requests. The requests received during that time are processed upon completion of the counting phase at that level. During each frequency counting level, a processor determines in which of these three phases it belongs to by keeping track of how many data-sequences it has read so. By comparing this number to the total number of data-sequences in its database it can determine whether or not it is at the beginning, middle, final stages of its computations.

If an idle processor does not receive work from a potential donor (either because the donor does not have work or in the middle of its counting phase), it chooses another processor to inquire about work. If after polling all available processors, an idle processor did not get any work, this processor selects an active processor with work and lets it know that it will wait for work until it gets it. To ensure that idle processors will choose different active processors, an idle processor will select top n (in terms of estimate of waiting time and remaining workload) active processors and will randomly choose one to attach itself to. Also, because each idle processor chooses a potential donor randomly, the donors are selected with equal probability, ensuring that work requests are evenly distributed.

6   Runtime and Scalability Analysis

The effectiveness of a particular parallel formulation can be analytically evaluated using a variety of metrics such as parallel runtime, speedup, efficiency, maximum degree of concurrency, and isoefficiency [7]. In general, a good

parallel formulation should minimize the overheads due to parallelization and maximize the number of processors that can utilize in a cost-effective manner. Moreover, a scalable parallel formulation should be able to maintain high parallel efficiency as the number of processors and the problem size that needs to be solved increases.

The goal of this section is to analytically evaluate and compare the data- and task-parallel formulations of the tree-projection algorithm. However, because the serial algorithm itself is quite complex and its runtime depends on the minimum support value and various characteristics of the input database (*i.e.*, the number of distinct items, the number of frequent subsequences that it contains, their length distribution, *etc*), our analysis will contain a number of simplifying assumptions and will be approximate. However, despite these limitations, we believe that it provides important information that allows us to compare and contrast these algorithms.

6.1 Data-Parallel Formulation

Since the depth of the projection tree depends on the value of the minimum support and the characteristics of the dataset, our analysis of the data-parallel formulation will focus on how effective it is in parallelizing the computations performed when the algorithm computes the frequency of the candidates corresponding to the $(k+2)$nd level of the tree. Since the overall algorithm performs a number of such computational steps one-after-the-other, knowing how effectively each step is parallelized is sufficient to determine the overall parallel performance and scalability of this algorithm.

Recall from Section 5.1 that the frequency of the candidates at level $k+2$ is computed by projecting the locally stored database sequences on the nodes of the kth level of the tree and computing the frequencies using four different matrices corresponding to different itemset and sequence extensions. The overall frequencies are determined by using a reduction operation to add these local frequencies. The dimensions of these matrices depend on the number of active items at each node of the tree, and in the worst case is upper bounded by $|I|$ (recall that I is the set of distinct items in the database). Thus, if m_k is the number of tree nodes at level k of the tree, then because the amount of data that is involved in the reduction operation (*i.e.*, $O(|I|^2 m_k)$) is much larger than the number of processors, this reduction can be done in time linear on the amount of data [7]; thus, the per-processor communication overhead T_k^o due to the reduction operation is

$$T_k^o = O(|I|^2 m_k). \tag{1}$$

The amount of time required by the serial algorithm T_k^s to compute the frequencies of the candidate patterns at level $k+2$ depends on the number of sequences in the database $|\mathcal{D}|$, the number of distinct items $|I|$, and the number of nodes m_k at level k, *i.e.*, $T_k^s = f(|\mathcal{D}|, |I|, m_k)$. Note that m_k depends both on $|I|$ and on the value of the minimum support σ. In particular, as $|I|$ increases and/or σ decreases, the number of frequent patterns will also increase and thus m_k will increase. Since each database sequence may end-up being projected to each node of the kth level, in the worst-case, $f(|\mathcal{D}|, |I|, m_k) = O(|\mathcal{D}||I|^2 m_k)$. However, this is only a loose upper bound because the projection is done in a recursive fashion that takes advantage of the tree structure. As a result, some of the work that is required to project the database on a particular node is shared with the work that is required by other nodes that have the same ancestor. For this reason, a more realistic bound for T_k^s is an expression of the form $O(|\mathcal{D}|^\alpha |I|^2 m_k)$, where $0 < \alpha < 1.0$. However, even though it is hard to estimate the value of α, the aggregate amount of work required to project a particular sequence to all the nodes at the kth level of the tree is the same for both the serial and the parallel formulation, *i.e.*, α will be the same for both the serial and parallel algorithm.

From the above discussion we can see that the amount of time required to project the database at level k and determine the frequency of the nodes at level $k+2$, T_k^p, on p processors is

$$T_k^p = \frac{T_k^s}{p} + T_k^o. \tag{2}$$

In order for this formulation to be cost-optimal, $T_k^o = O(T_k^s/p)$, which indicates that the maximum number of processors that the DPF algorithm can use cost-optimally is given by

$$p = O\left(\frac{T_k^s}{|I|^2 m_k}\right).$$

Assuming that the overall dataset characteristics remain them same, the amount of work performed by the serial algorithm can increase for two different reasons. First, it can be due to an increase in the number of input sequences $|\mathcal{D}|$. Second, it can be due to a decrease in the value of the minimum support σ. If $|\mathcal{D}|$ increases, then because T_k^s also increases as a function of $|\mathcal{D}|$, the number of processors that can be used cost-effectively by the DPF algorithm also increases. However, if σ decreases, then since more frequent patterns will be discovered, both the number of nodes m_k in the tree and T_k^s (as it depends on m_k) will tend to increase. Moreover, these two quantities will increase at roughly the same rate. Thus, even though the work has increased, the DPF algorithm cannot use more processors without a decrease in the parallel efficiency.

Besides the above limitation, the DPF algorithm works well only when the tree and count matrices can fit in the main memory of each processor. If the number of candidates is large, then the matrices may not fit in the main memory. In this case, this algorithm has to partition the tree and compute the counts by scanning the database multiple times, once for each partition of the tree.

6.2   Static Task-Parallel Formulation

The parallel performance of the static task-parallel formulation of the tree-projection algorithm depends on two parameters. First, is the degree to which the static assignment of the tasks to processors is capable of equally balancing the work and thus eliminating any overheads due to processor idling. Second, is the amount of time required to actually perform the task decomposition itself and redistribute the database among the processors. As discussed in Section 5.2, the effectiveness of the static task-parallel formulation in minimizing idling overheads is dataset dependent and as such very hard to evaluate analytically. For this reason our analysis will primarily focus on determining the overhead due to task distribution.

Recall from Section 5.2.1 that the task distribution takes place when the projection tree has been grown up to level $k+1$ using the DPF algorithm by distributing the nodes of the kth level of the tree. Assuming that we use the bipartite graph-partitioning-based approach, this is achieved by computing a p-way partitioning of the bipartite graph that contains $m_k + |I|$ vertices. If $m_k + |I| >> p$, this partitioning can be computed in $O(m_k + |I|)$ time [12].

Once the node/task assignment has been determined, the algorithm proceeds to perform the actual redistribution of the database so that each processor can obtain and store locally the portion of the database that it needs to further grow the subtrees rooted at its assigned nodes. The communication time spent during this redistribution depends on the size of the database that each processor needs to receive. In the ideal case, the task assignment will result in minimal overlap in the portions of the database assigned to each processor. Assuming that the underlying parallel computer has sufficient cross-bisection bandwidth [7], then in the ideal case, the communication time is $O(|\mathcal{D}|/p)$, since each processor will receive, on the average, $1/p$th of the database. However, depending on the amount of overlap, the actual communication time may be higher. For the purpose of this analysis, we will assume that $\beta|\mathcal{D}|/p$ (where $\beta \geq 1$) is the average number of database sequences that need to be received by each processor. Then, the amount of time spent during task distribution is

$$T^{dist} = O(m_k + |I|) + O\left(\beta \frac{|\mathcal{D}|}{p}\right). \tag{3}$$

The time T_i^{local} required by each processor P_i to completely grow the subtrees rooted at the nodes that it was assigned depends on the size of its local database $\beta|\mathcal{D}|/p$, and the total number of descendant nodes d_i that will be generated. Using the notation introduced in Section 6.1, we have that

$$T_i^{local} = f\left(\beta \frac{|\mathcal{D}|}{p}, |I|, d_i\right) = O\left(\left(\beta \frac{|\mathcal{D}|}{p}\right)^{\alpha} |I|^2 d_i\right). \tag{4}$$

The task-parallel formulation will be effective if the time required to perform the task decomposition (T^{dist}) is substantially smaller than the time required by the processors to grow the subtrees assigned to them (T_i^{local}). In general this will be true as long as $d_i >> m_k$, $i.e.$, the number of nodes that each processor will end up generating is greater than the number of nodes at the level in which the task decomposition took place.

Furthermore, from Equations 3 and 4 we have that the maximum number of processors that can be used in a

cost-effective manner is

$$p = O\left(\frac{\beta|\mathcal{D}|d_i^{1/\alpha}|I|^{2/\alpha}}{m_k^{1/\alpha}}\right).$$

Thus, in terms of scalability, if the overall work increases by increasing $|\mathcal{D}|$, then the task-parallel formulation can use more processors in a cost-effective manner. If the overall work increases as a result of a reduction of the minimum support value, then because the resulting tree will be deeper and have more nodes, the ratio d_i/m_k will also increase and thus the task-parallel formulation will be able to utilize more processors in a cost-effective manner, as well. Thus, the task-parallel formulation is more scalable than the data-parallel one.

Moreover, the task-parallel formulation has one additional advantage, which is that it can potentially lead to super-linear speedup. This is because along with a partitioning of the tree nodes it also reduces the size of the database that needs to be accessed while growing the subtrees underneath these nodes. As a result, each processor not only it has to process fewer nodes (which is the source of concurrency that this formulation exploits) but it also spends less time per node compared to the serial algorithm. The runtime savings depend directly on how effective the node distribution methodology is in minimizing the database overlap across the different processors. Note that this super-linearity suggests that the serial algorithm itself can be modified to yield better performance. In fact this has been recognized by a number of researchers, and a number of serial algorithms have been developed that take advantage of it [21, 24].

7 Experimental Evaluation

We evaluated the various parallel formulations for the sequential tree-projection algorithm on a 32-processor IBM SP cluster consisting of eight quad-CPU nodes connected via a high-speed switch. Each node had 4GB of memory and the CPUs were based on the Power3 architecture running at 222MHz. All algorithms were implemented using the message-passing programming paradigm and we used MPI [15] as the communication library. Moreover, for the reasons discussed in Section 5, our parallel implementations assume that the database reside on the disk.

These algorithms were evaluated on four different datasets DS1, DS2, DS3, and DS4 that were generated using the synthetic dataset generated provided by the IBM Quest group [27]. Each of these datasets contained one million sequences and were generated by setting the number of distinct items to 10,000, the number of maximally potentially frequent sequences to 5,000, the number of maximal potentially frequent itemsets to 25,000, and the average length of maximal potentially frequent sequences to four. Depending on the dataset the average number of itemsets per sequence ranged from 10 to 20, the average number of items per itemset ranged from 2.5 to 5, and the average length of the itemsets in maximal potentially frequent sequences ranged from 1.25 to 2.5. The individual set of parameters used to generate each dataset is shown in Table 1.

In addition, for each dataset, we used a different minimum support value, which was selected so that the resulting problem instance will be both non-trivial and also finish in a reasonable amount of time. These minimum support values are shown at the last row of Table 1.

Parameter	Dataset			
	DS1	DS2	DS3	DS4
Database Size	1,000,000	1,000,000	1,000,000	1,000,000
No. of Distinct Items	10,000	10,000	10,000	10,000
No. of Maximal Potentially Frequent Sequences	5,000	5,000	5,000	5,000
No. of Maximal Potentially Frequent Itemsets	25,000	25,000	25,000	25,000
Avg. Length of Maximal Potentially Frequent Sequences	4	4	4	4
Avg. No. of Itemsets per Sequence	10	10	10	20
Avg. No. of Items per Itemset	2.5	5	5	2.5
Avg. Length of Itemsets in Maximal Potentially Frequent Sequences	1.25	1.25	2.5	1.25
Minimum Support (%)	0.1	0.25	0.33	0.25

Table 1: The parameters of the data-generator that were used to obtain the various datasets.

7.1 Evaluation of Static Parallel Formulations

Our first set of experiments was designed to evaluate and compare the performance of the data-parallel and the static task-parallel formulations. Toward this goal we performed a series of experiments in which we found the frequent sequential patterns of all four datasets on 2, 4, 8, 16, and 32 processors using the data-parallel algorithm and four variations of the static task-parallel algorithm. These four variations were derived by using both the bin-packing and the bipartite graph-partitioning based approaches to balance the work, and by expanding the projection tree up to levels two and three before switching to the task-parallel formulation.

Dataset DS1										
	DPF		STPF-BP2		STPF-BP3		STPF-GP2		STPF-GP3	
Processors	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
2	2042.63	1.00	1883.79	1.00	2473.94	1.00	1689.95	1.00	1763.29	1.00
4	1042.33	1.96	861.23	2.19	1514.07	1.63	676.83	2.50	732.23	2.41
8	529.50	3.86	397.41	4.74	810.02	3.05	255.21	6.62	315.69	5.59
16	293.82	6.95	187.83	10.03	413.13	5.99	109.29	15.46	144.91	12.17
32	166.21	12.29	105.92	17.79	243.49	10.16	56.70	29.81	73.09	24.12

Dataset DS2										
	DPF		STPF-BP2		STPF-BP3		STPF-GP2		STPF-GP3	
Processors	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
2	3514.25	1.00	3313.56	1.00	4270.45	1.00	2933.57	1.00	2890.84	1.0.
4	1786.44	1.97	1506.92	2.20	2422.64	1.76	1224.34	2.40	1249.35	2.31
8	921.43	3.81	776.94	4.26	1263.66	3.38	445.42	6.59	494.96	5.84
16	477.13	7.37	317.70	10.43	636.65	6.71	241.60	12.14	235.56	12.27
32	252.25	13.93	192.08	17.25	304.93	14.00	119.72	24.50	126.11	22.92

Dataset DS3										
	DPF		STPF-BP2		STPF-BP3		STPF-GP2		STPF-GP3	
Processors	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
2	4026.08	1.00	3583.78	1.00	3637.16	1.00	2516.28	1.00	2489.08	1.00
4	2028.30	1.98	1797.55	1.99	1864.87	1.95	1035.96	2.43	1087.02	2.29
8	1058.22	3.80	832.84	4.30	999.20	3.64	901.38	2.79	561.00	4.44
16	551.97	7.29	338.63	10.58	492.14	7.39	396.00	6.35	363.00	6.86
32	300.50	13.40	251.78	14.23	237.92	15.28	314.37	8.00	291.09	8.55

Dataset DS4										
	DPF		STPF-BP2		STPF-BP3		STPF-GP2		STPF-GP3	
Processors	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
2	2686.35	1.00	2554.17	1.00	3088.33	1.00	2169.73	1.00	2257.55	1.00
4	1376.76	1.95	1205.20	2.12	1878.34	1.64	983.75	2.21	998.36	2.26
8	733.83	3.66	602.25	4.24	970.45	3.18	389.54	5.57	423.95	5.33
16	358.42	7.50	285.85	8.93	526.64	5.86	184.70	11.75	197.83	11.41
32	197.58	13.60	154.73	16.51	275.05	11.23	147.95	14.67	124.95	18.07

Table 2: The amount of time (in seconds) required by the static parallel formulations on the various datasets and the speedup that was achieved relative to the two-processor runs.

Table 2 shows the results obtained by this series of experiments. The columns labeled "DPF" show the results obtained by the data-parallel formulation. The columns labeled "STPF-BP2" and "STPF-BP3" show the results obtained by the bin-packing-based static task-parallel formulation when the projection tree was initially grown up to levels two and three, respectively. The columns labeled "STPF-GP2" and "STPF-GP3" show the corresponding results of the scheme that uses the bipartite graph-partitioning approach to split the work among the processors. For each scheme, Table 2 shows the amount of time (in seconds) that was required and the speedup that was achieved relative to the amount of time required by the corresponding two-processor result. For example, for DS1 and DPF, the relative speedup on eight processors was derived by diving the two-processor runtime of the DPF scheme (2042.63) with the eight-processor runtime (529.50).

DPF Performance   From the results of Table 2 we can see that for all four dataset the DPF algorithm achieves good performance and its parallel runtime decreases substantially as we increase the number of processors. In particular,

comparing the amount of time required by two and 32 processors (a factor of 16), we can see that it decreases by a factor of 12.29, 13.93, 13.40 and 13.60 for each one of the four datasets, respectively. Note that as expected, because for each dataset the problem size was kept constant, the parallel efficiency of the algorithm decreases as we increase the number of processors. This is because the amount of work performed by each processor decreases whereas the communication overhead due to the reduction operation (Equation 1) remains constant.

STPF Performance   The results of Table 2 show that the various instances of the static task-parallel algorithm lead to dramatically different performance. In particular, comparing the bipartite graph-partitioning-based formulations of STPF to those based on bin-packing we can see that the former lead to smaller execution times. Moreover, the relative gap increases as the number of processors increases. For example, for DS1, STPF-GP3 is about 1.4 times faster than STPF-BP3 on two processors, whereas it is 3.3 times faster on 32 processors. This is because compared to the bin-packing-based approach, the bipartite graph-partitioning-based approach reduces the overlap among the local databases (as discussed in Section 5.2) and leads to a task distribution that reduces the average size of the database that is required by each processor. This directly reduces both the amount of time required to initially redistribute the database and the amount of time required to project these databases during the follow up computations.

(a) DS1

(a) DS2

(a) DS3

(a) DS4
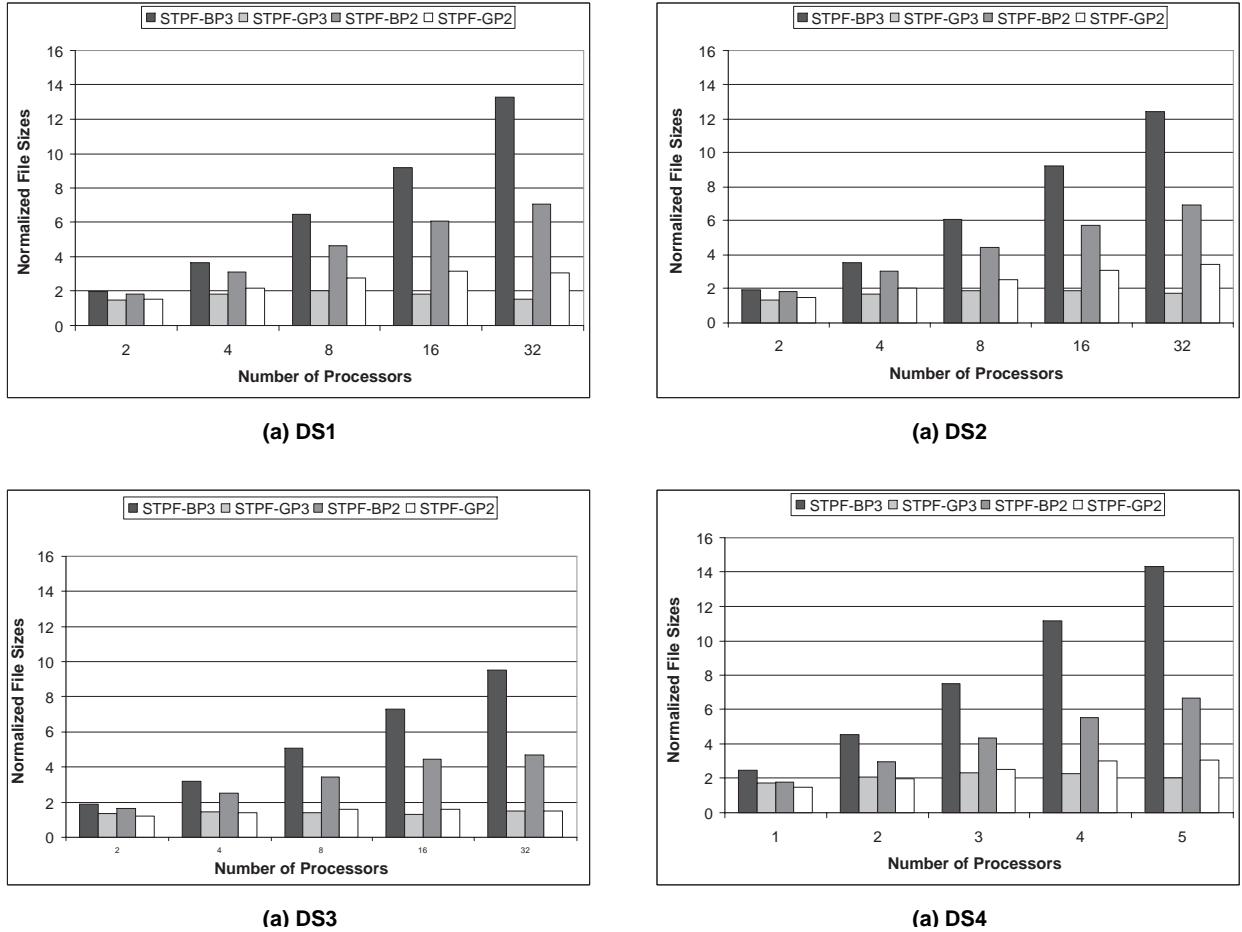
Figure 4: The aggregate size of the local databases required by the bin-packing and bipartite graph-partitioning-based approaches relative to the size of the original database.

To illustrate the reduction in the size of the databases resulting by using the bipartite graph-partitioning-based approach over that based on bin-packing, we plotted the size of the local databases (summed over all the processors) for these two schemes. These results are shown in Figure 4. These figures show the size of the databases relative to the corresponding database-size on a single processor. Note that for all schemes, the local databases have been

17

Processors	DPF	STPF-BP2	STPF-BP3	STPF-GP2	STPF-GP3
2	7914.82	6128.41	6885.33	4370.37	4263.24
4	7922.66	4835.05	6740.49	3012.61	3443.54
8	8017.73	3626.02	6294.13	2346.55	2785.41
16	8135.03	2903.85	5508.26	1992.70	2595.51
32	8413.89	2404.73	4604.70	1804.09	2497.52

Table 3: Serial execution time for DS3

pruned so that they contain only the active items for each processor. As we can see from these figures, the graph-partitioning-based schemes lead to local databases whose size is up to an order of magnitude smaller that that of the bin-packing-based scheme. In addition, as shown in Figure 4 for most of the graph-partitioning-based problem instances, the extra storage that is required due to overlapping databases is at most twice that of the original database.

The results in Table 2 also confirm the discussion in Section 6.2, which predicted that the static task-parallel formulations can potentially achieve super-linear speedups. Such super-linearity is observed for both the bin-packing and graph-partitioning-based approaches, even though they are more pronounced for the latter. For instance, for STPF-GP2, as the number of processors increases from 2 to 32, the runtime for each one of the four datasets decreases by factor of 29.81, 24.50, 8.0 and 14.67, respectively. Note that the relatively poor results for DS3 is due to the fact that the static tasks assignment leads to unbalanced work distribution.

As discussed in Section 6.2, the super-linear results is due to the fact that the STPF algorithm changes key characteristics of the serial tree-projection algorithm which results in an overall reduction in the amount of time spent by all processors in performing *essential* computations (*i.e.*, computations that will be performed by the serial algorithm and are not associated with parallelization). To verify this observation, we instrumented our programs to record the total amount of time spent by the processors in essential computations. Table 3 reports those times for DS3. In this table, the times reported for DPF are the most closely related to the actual serial execution time of the tree-projection algorithm and as we can see they increase slowly as we increase the number of processors. This increase is primarily due to the fact that as p increases the times reported in Table 3 contain (i) various house-keeping operations performed by all processors (*e.g.*, memory allocation/de-allocation, array initialization, *etc.*) and (ii) contention for disk I/O[1]. However, the amount of time spent on essential computations by the various task-parallel formulations is smaller than that of DPF. Moreover, this time reduces as the number of processors increases, which is the reason for the super-linear speedups. Also note that the essential computational time required by the graph-partitioning-based formulations is smaller than that required by bin-packing as it leads to smaller local databases.

Finally, comparing STPF-BP2 versus STPF-BP3 and STPF-GP2 against STPF-GP3, we can see that the run-times achieved by the schemes that switch to tasks distribution after the second level are in general smaller. The only exception is for DS3 in which STPF-GP3 does better that STPF-GP2 on 8, 16 and 32 processors. This is due to the fact that STPF-GP3 leads to better load-balance and that STPF-GP2 is highly unbalanced.

7.2 Evaluation of Dynamic Task Parallel Formulation

Our second set of experiments was focused on evaluating the effectiveness of the dynamic load-balancing scheme for the task-parallel formulation (DTPF). Toward this goal we performed a series of experiments in which the work was initially distributed using the STPF-GP2 scheme and then during the computations it was was dynamically load-balanced using the scheme described in Section 5.2.2. The STPF-GP2 scheme was selected because, as the results in Table 2 showed, it consistently outperformed the other static task-parallel schemes. Table 4 shows the amount of time and the speedup (relative to the two-processor results) achieved by various instances of the dynamic task-parallel algorithm on the four datasets on 2, 4, 8, 16, and 32 processors. In addition, in order to make comparisons easier, Table 4 also shows the results achieved by the STPF-GP2 algorithm, which are the same with those presented in the corresponding columns of Table 2.

Recall from Section 5.2.2, that the dynamic load-balancing algorithm divides the counting phase of each processor into three stages, the early, intermediate, and late, and uses a different protocol for handling work requests in each

---

[1]The underlying physical parallel system on which the experiments were performed consisted of SMP nodes and each processor had to compete with the other processors in that node for file-system operations.

of these stages. The results in Table 4 show three different instances of DTPF. The first instance DTPF-25-50-25 corresponds to an algorithm in which each one of the early and late stages represent approximately 25% of the overall computation (with respect to a particular counting phase), and the remaining 50% correspond to the intermediate stage. Similarly, the results labeled "DTPF-25-70-5" and "DTPF-0-75-25" correspond to instances of the algorithm with a 25%, 70%, 5%, and a 0%, 75%, 25% division between early, intermediate, and late stages, respectively.

Dataset DS1								
	STPF-GP2		DTPF-25-50-25		DTPF-25-70-5		DTPF-0-75-25	
Processor	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
2	1689.95	1.00	1746.43	1.00	1644.66	1.00	1635.09	1.00
4	676.83	2.50	643.13	2.72	663.66	2.48	648.83	2.52
8	255.21	6.62	236.66	7.38	247.36	6.65	242.51	6.74
16	109.29	15.46	111.42	15.67	110.99	14.83	107.13	15.26
32	56.70	29.81	60.10	29.06	63.36	25.83	62.34	26.23

Dataset DS2								
	STPF-GP2		DTPF-25-50-25		DTPF-25-70-5		DTPF-0-75-25	
Processor	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
2	2933.57	1.00	3025.35	1.00	2884.10	1.00	2774.06	1.00
4	1224.34	2.40	1124.99	2.70	1239.56	2.33	1131.99	2.45
8	445.42	6.59	448.24	6.75	477.06	6.05	450.78	6.15
16	241.60	12.14	201.90	14.98	215.66	13.37	231.89	11.96
32	119.72	24.50	119.01	25.42	122.32	23.58	128.17	21.64

Dataset DS3								
	STPF-GP2		DTPF-25-50-25		DTPF-25-70-5		DTPF-0-75-25	
Processor	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
2	2516.28	1.00	2386.42	1.00	2339.03	1.00	2342.18	1.00
4	1035.96	2.43	876.37	2.72	901.79	2.59	840.73	2.79
8	901.38	2.79	391.35	6.10	431.80	5.42	388.76	6.02
16	396.00	6.35	193.07	12.36	216.12	10.82	189.75	12.34
32	314.37	8.00	116.62	20.46	135.43	17.27	129.37	18.10

Dataset DS4								
	STPF-GP2		DTPF-25-50-25		DTPF-25-70-5		DTPF-0-75-25	
Processor	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
2	2169.73	1.00	2248.32	1.00	2180.04	1.00	2183.50	1.00
4	983.75	2.21	857.14	2.63	918.62	2.37	828.06	2.64
8	389.54	5.57	353.23	6.37	388.25	5.62	344.22	6.34
16	184.70	11.75	165.47	13.59	179.84	12.12	185.86	11.75
32	147.95	14.67	100.92	22.28	106.30	20.51	107.06	20.40

Table 4: The amount of time (in seconds) required by the task-parallel formulations that use dynamic load-balancing and the speedup that was achieved relative to the two-processor runs.

A number of observations can be made by looking at these results. First, comparing the various instances of DTPF against STPF-GP2 we can see that for each one of the different experiments DTPF achieves comparable or substantially lower run-times. In particular, focusing on the 32-processor results we can see that for DS1 and DS2, DTPF is comparable to STPF and for DS3 and DS4, DTPF-25-50-25 is 2.7 and 1.5 times faster, respectively. The dramatic improvement on the DS3 is due to the fact that the underlying tree leads to highly unbalanced computations, which the DTPF can balance by dynamically moving work between the processors. To better illustrate this we calculated the workload imbalance of STPF-GP2 for all four datasets. These results are shown in Table 5. From the results in this table we can see that the STPF algorithm produces highly unbalanced computations for DS3 regardless of the number of processors. However, for the remaining data sets, the overall computations are not significantly unbalanced, and for this reason the gains achieved by DTPF are much smaller. Moreover, in some cases the overhead incurred by DTPF (due to its dynamic load-balancing protocol and termination detection algorithm) can lead to slight increases in the overall parallel execution time.

Second, comparing the different instances of the DTPF algorithm we can see that for small number of processors, DTPF-0-75-25 does somewhat better compared to DTPF-25-50-25. However, as number of processors increases,

	DS1					DS2				
Processors	2	4	8	16	32	2	4	8	16	32
Load Imbalance	1.03	1.07	1.05	1.09	1.23	1.04	1.09	1.07	1.21	1.35
	DS3					DS4				
Processors	2	4	8	16	32	2	4	8	16	32
Load Imbalance	1.13	1.31	2.83	2.32	4.11	1.01	1.03	1.11	1.23	1.22

Table 5: Load imbalance of the STPF-GP2 approach.

DTPF-25-50-25 achieves lower run-times. In particular on 32 processors, DTP-25-50-25 is about 4%-12% faster. The reason for this is that for small number of processors, the overall number of work transfer requests is quite small, and can arrive within the small time window between successive counting phases. However, as the number of processors increases, the large time window of the early stage allows the algorithm to reduce idling overheads.

8 Conclusions

In this paper we presented three different algorithms for finding sequential patterns using the tree-projection algorithm that are suitable for distributed-memory parallel computers. Our experimental results showed that both the data- and the task-parallel formulations are able to achieve good speedups as the number of processors increase. Furthermore, the bipartite graph-partitioning-based task distribution approach is able to substantially reduce the overlap in the databases required by each processor. However, as number of processors increases, the accuracy by which the work-load can be estimated decreases and the computation became increasingly un-balanced. To overcome this problem, we developed a new dynamic load-balancing algorithm, which was able to achieve good speedups as the number of processors increases. Furthermore, the overall performance is improved in comparison to static load-balancing algorithm.

Even though the algorithms in this paper were developed in the context of a particular projection-based algorithm, other projection-based algorithms [21, 24] can be parallelized in a similar fashion. We expect that the various parallel formulations that we developed to exhibit similar performance for these algorithms as well.

References

[1] R.C. Agarwall, C. Aggarwal, and V.V.V. Prasad. A tree projection algorithm for generation of frequent itemsets. *Journal of Parallel and Distributed Computing (Special Issue on High Performance Data Mining)*, 2000.

[2] R. Agrawal and J.C. Shafer. Parallel mining of association rules. *IEEE Transactions on Knowledge and Data Eng.*, 8(6):962–969, December 1996.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of the 20th VLDB Conference*, pages 487–499, Santiago, Chile, 1994.

[4] S. Brin, R. Motwani, and C. Silversteim. Beyond market baskets: Generalizing association rules to correlations. In *Proc. of 1997 ACM-SIGMOD Int. Conf. on Management of Data*, Tucson, Arizona, 1997.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, McGraw-Hill, New York, NY, 1990.

[6] E. W. Dijkstra, W. H. Seijen, and A. J. M. Van Gasteren. Derivation of a termination detection algorithm for a distributed computation. *Information Processing Letters*, 16–5:217–219, 1983.

[7] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing: Design and Analysis of Algorithms, 2nd Edition*. Adison Wesley Publishing Company, Redwood City, CA, 2003.

[8] E.H. Han, G. Karypis, and V. Kumar. Scalable parallel data mining for association rules. *IEEE Transactions on Knowledge and Data Eng.*, 12(3):337–352, 2000.

[9] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. of 2000 ACM-SIGMOD Int. Conf. on Management of Data*, 2000.

[10] Mahesh V. Joshi, George Karypis, and Vipin Kumar. Universal formulation of sequential patterns. Technical report, Universit of Minnesota, Department of Computer Science, Minneapolis, 1999.

[11] G. Karypis and V. Kumar. METIS: Unstructured graph partitioning and sparse matrix ordering system. Technical report, Department of Computer Science, University of Minnesota, 1995. Available on the WWW at URL *http://www.cs.umn.edu/˜karypis/metis.*

[12] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. In *Proceedings of Supercomputing*, 1998. Also available on WWW at URL http://www.cs.umn.edu/˜karypis.

[13] George Karypis and Vipin Kumar. Unstructured tree search on simd parallel computers. *Journal of Parallel and Distributed Computing*, 22(3):379–391, September 1994.

[14] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovering frequent episodes in sequences. In *Proc. of the First Int'l Conference on Knowledge Discovery and Data Mining*, pages 210–215, Montreal, Quebec, 1995.

[15] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, May 1994. Available at http://www.mpi-forum.org.

[16] Andreas Mueller. Fast sequential and parallel algorithms for association rule mining: A comparison. Technical Report CS-TR-3515, College Park, MD, 1995.

[17] J. Park, M. Chen, and P. Yu. An efficient parallel data mining for association rules. In *Proceedings of the 4th International Conference on Information and Knowledge Management*, 1995.

[18] J.S. Park, M.S. Chen, and P.S. Yu. An effective hash-based algorithm for mining association rules. In *Proc. of 1995 ACM-SIGMOD Int. Conf. on Management of Data*, 1995.

[19] J.S. Park, M.S. Chen, and P.S. Yu. Efficient parallel data mining for association rules. In *Proceedings of the 4th Int'l Conf. on Information and Knowledge Management*, 1995.

[20] S. Parthasarathy, M. Zaki, M. Ogihara, and W. Li. Parallel data mining for association rules on shared-memory systems. *Knowledge and Information Systems*, 3(1):1–29, 2001.

[21] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Helen Ping, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proceedings 2001 International Conference on Data Engineering*, 2001.

[22] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. of the 21st VLDB Conference*, pages 432–443, Zurich, Switzerland, 1995.

[23] Masakazu Seno and George Karypis. Lpminer: An algorithm for finding frequent itemsets using length-decreasing support constraint. In *IEEE International Conference on Data Mining*, 2001. Also available as a UMN-CS technical report, TR# 01-026.

[24] Masakazu Seno and George Karypis. Slpminer: An algorithm for finding frequent sequential patterns using length-decreasing support constraint. In *IEEE International Conference on Data Mining*, 2002.

[25] Takahiko Shintani and Masaru Kitsuregawa. Hash based parallel algorithms for mining association rules. In *Proc. of the Conference on Paralellel and Distributed Information Systems*, 1996.

[26] T. Shitani and M. Kisuregawa. Mining algorithms for sequential patterns in parallel: Hash based approach. In *Pacific–Asia Conference on Knowledge Discovery and Data Mining*, 1998.

[27] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proc. of the Fifth Int'l Conference on Extending Database Technology*, Avignon, France, 1996.

[28] O. Zaiane, M. El-Hajj, and P. Lu. Fast parallel association rule mining without candidacy generation. In *IEEE International Conference on Data Mining*, pages 665–668, 2001.

[29] M. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency*, 7(4):14–25, 1999.

[30] Mohammed J. Zaki. Parallel sequence mining on shared-memory machines. In Vipin Kumar, Sanjay Ranka, and Vineet Singh, editors, *Journal of Parallel and Distributed Computing, special issue on High Performance Data Mining*, volume 61, pages 401–426, 2001.

[31] Mohammed J. Zaki. SPADE: an efficient algorithms for mining frequent sequences. *Machine Learning Journal*, 42:31–60, 2001.

[32] Mohammed Javeed Zaki. Scalable algorithms for association mining. *Knowledge and Data Engineering*, 12(2):372–390, 2000.