Parallel Graph Mining on Shared Memory Architectures[*]

Gregory Buehrer and Srinivasan Parthasarathy[†]
Department of Computer Science and Engineering,
The Ohio State University, Columbus, OH 43210, USA

Anthony Nguyen, Daehyun Kim, Yen-Kuang Chen, and Pradeep Dubey
Applications Research Laboratory,
Intel Corporation, Santa Clara, CA 95052, USA

Abstract

Mining graph based data sets has practical applications in many areas including molecular substructure discovery, web link analysis, fraud detection, and social network analysis. The discovery challenge in graph mining is to find all subgraphs which occur in at least σ graphs of a graph database, where σ is user specified parameter. Subgraph isomorphism, the enormous search space of candidate graph patterns, and the importance of interactive response times make addressing this challenge particularly daunting.

In this work, building on the existing state-of-the-art, we propose a novel approach for parallelizing such algorithms on shared memory multiprocessor systems. We present several novel task partitioning and load balancing schemes, and evaluate their efficacy. We also propose several queuing models which afford dynamic task sharing. We show that dynamic partitioning and dynamic task allocation provide a synergy which greatly improves scalability. Our parallelization algorithm achieves excellent speedup of 27 times on 32 nodes for several real world datasets, as compared to a naive approach which affords only 5-fold speedup. We also discuss implications of this work in light of recent trends in micro-architecture design, particularly multiple core and multithreaded systems.

1 Introduction

Mining frequent structures in graphical data sets is an important problem with applications in a broad range of domains. Finding frequent patterns in graph databases such as chemical and biological data [23], XML documents [22], web link data [19], financial transaction data [12], social network data[20], protein folding data [30], and other areas has posed a nice challenge to the data mining community. Representing these data sets in graphical form can capture relationships between entities which are difficult or inefficient when captured in relational tables. For example, financial transactions can be stored as graphs, where each node is an institution and the edge is an

[*]Contact emails – [buehrer,srini]@cse.ohio-state.edu

[†]This work is supported in part by NSF grants (#CAREER-IIS-0347662), (#RI-CNS-0403342), and (#NGS-CNS-0406386).

account number. Also, molecular data is easily represented in graphical form. Finally, web logs can be mapped to user sessions, and then represented as graphs [19].

Frequent pattern mining was first introduced by Agrawal et al [1] in 1994, and has been extensively [2, 9, 11, 15, 17, 23, 25]. Frequent substructure mining, or graph mining, is the process of enumerating all subgraphs which occur in at least σ graphs in a graph database. σ is called the support threshold, and is typically supplied by the user.

In web session data, frequent patterns represent resources which are more often used, and the paths most often taken to access these resources. Application service providers can improve web server behavior by prefetching resources as users move the web application according to their typical session patterns. In molecular data sets, nodes correspond to atoms, and edges represent chemical bonds. Frequent substructures may provide insight into the behavior of the molecule. For example, graphs containing benzene rings connected to H-C=O groups tend to have a pleasant odor. Frequent substructure mining can be used to group previously unknown groups, affording new information. Frequent substructure mining can be used to group previously unknown groups, affording new information. Zaki et al [30] apply graph-based methods on weighted secondary graph structures of proteins to predict folding sequences. Finally, in our financial transaction data set, large graphs with high occurrence rates could represent potential criminal activity, such as money laundering.

One challenge in pattern mining is that the search space is exponential with respect to the data set, forcing runtimes to be quite long. Mining frequent graphs is made even more difficult due to the sub-graph isomorphism challenge. A simple chemical data set of 300 molecules can require many hours to mine when the support is set sufficiently low. Parallel architectures can improve these execution times when supplied with a scalable parallel program. At a high level, two parallel architectures are available; distributed clusters and shared memory machines. The increase in processor frequencies over the past several years has required a significant increase in voltage (and thus wattage), which has increased power consumption and heat dissipation of the central processing unit. In addition, increased frequencies require considerable extensions to instruction pipeline depths. To this end, chip manufacturers are moving from single CPUs with extremely high frequencies to lower frequency chips with multiple processing cores[1]. Even clusters employing message passing interfaces will likely be clusters of inexpensive, highly parallel shared memory machines. Such trends in micro-architecture design have motivated us to develop a graph mining algorithm for shared memory systems.

Thus, the key contributions of this paper are

- we provide a highly scalable parallel graph mining algorithm
- we evaluate several task partitioning models, and show that a dynamic model affords the lowest runtimes
- we evaluate several queuing models, showing that distributed queuing provides the best balance between temporal locality, concurrency, and locking costs when sharing tasks

[1]http://www.intel.com/cd/ids/developer/asmo-na/eng/201969.htm?page=6

2 Related Work

Advantages for representing data as graphs has been widely studied [5, 6, 11, 21, 27, 30]. Several serial algorithms exist which enumerate frequent subgraphs. FSG [11] mines for all frequent subgraphs using an APRIORI breadth-first fashion. Efficiency is gained by storing intermediate representations, and using vertex invariants to develop a canonical labeling system.

GSpan [25] uses a combination of depth-first and breadth-first trajectories to discover frequent patterns. Unlike previous methods, it only grows graphs with candidates existing in the data set. For canonical labeling, the authors define a mechanism called DFS coding for labeling graphs. This new labeling system reduced the search space considerably. It produced a significant reduction in execution time over previous methods. In addition, gSpan consumes relatively little memory.

Another recent serial graph miner is Gaston [15], developed by Nijssen and Kok. Gaston is an excellent depth-first breadth-first miner as well. It incorporates embedding lists using an efficient pointer-based structure. This provides an improvement in execution time at the cost of significantly increased memory consumption. A second contribution of the work is that they illustrate partitioning the search space into three subareas (paths, trees and cyclic graphs), an additional performance improvement is possible. In addition, the authors point out that for directed acyclic graphs, a candidate can be evaluated in constant time to determine if its incorporation into the current graph will result in a canonical label. The authors build on depth first traversals to specialize labeling systems for all three structures. They optionally employ the Naughty-McKay [14] for normalizing cyclic graphs. However, a key limitation is that it does not lend itself to parallelization because of the parent-child dependencies created with embedding lists.

Meinl et al [24] independently analyzed four serial graph mining algorithms, namely MoFa[3], FFSM[10], Gaston and gSpan. They conclude (in serial execution) that a) item Gaston and gSpan have the lowest runtimes, b) gSpan uses the least memory, c) efficient candidate generation (how to extend the current graph) is vital, d) embedding lists can reduce runtimes but incur extensive memory consumption, and e) embedding lists are not required to achieve high performance.

Wang and Parthasarathy [23] developed a parallel algorithm for their Motif Miner toolkit [17]. Motif Miner searches for interesting substructures in noisy geometric graphs (or graphs embedded in a 3-dimensional space) targeted at large biochemical molecules such as proteins. However, their parallelization strategy cannot be directly applied to the more general graph mining problem. Cook et al [5] have been developing a system called Subdue for several years. Their research has produced a parallel implementation, which solves a somewhat different problem, using approximations and user interaction to return potentially interesting substructures again on geometric graphs.

There have been several approaches for parallel mining of other frequent patterns [18, 29]. Strategies for mining associations in parallel have been proposed by various researchers. Zaki [28] proposed parallel partitioning schemes for sequence mining. The author illustrates the benefits of dynamic load balancing in shared memory environments. However, a key difference is that estimates for task execution time (used by the author) are far easier to compute in sequence mining than for graph mining. Guralnik and Karypis had similar findings in [8].

In addition to the above differences, the sequence mining domain and especially the association mining domain has a search space that is typically much smaller than that faced by graph mining algorithms. Moreover, the sub-graph isomorphism problem adds a further wrinkle to the problem.

An additional burden to parallelism is that interconnections between nodes in a graph makes it particularly difficult to split up work in a totally independent fashion.

Our own efforts in sequence and itemset mining have shown us that static load balancing models behave poorly in pattern mining because the time to mine even a single task is highly variable [16]. We expect graph mining to exhibit worse behavior. Some researchers [4] have used random sampling to combat this problem for other frequent pattern algorithms, although it has not been shown to be effective for graph mining, since random sampling may not capture important relations among nodes.

3 Parallel Graph Mining Algorithm

The task of frequent graph mining is to enumerate all subgraphs which appear in at least σ graphs in the data set, where σ is a user-defined threshold called *minimum support*. As discussed in Section 2, there are several serial graph mining algorithms we may use as a starting point for parallelization. Our parallel algorithm employs many of the techniques of gSpan [25], which has been shown to be one of the fastest and most efficient graph mining algorithms [24]. Further justification for this choice follows later in this section.

3.1 Problem Statement

Our definition of the problem is unchanged from [11, 15, 10, 25]. A graph is a set of vertices V and a set of edges $E \subseteq (V \times V)$, where V is unrestricted (graphs can have cycles). Nodes and edges may have labels, and may be directed or undirected. Let us define the labeling function $l : (E, V) \rightarrow L$, where all elements of E and V are mapped to exactly one label. A graph g' is a subgraph of g if there is a bijection from the edges $E' \subseteq E$ with matching labels for nodes and edges. This mapping is also called an embedding of g' in g. If both g' and g are subgraphs of each other, they are considered isomorphic. A subgraph g' is frequent if it there is an embedding in at least σ graphs in the database, where σ is a user-defined threshold, or $\#(g \mid g \in G \mid g' \subset g)$. The problem is then to enumerate all frequent graphs g in D.

3.2 Search Space Traversal

We process the search space in depth-first order. It allows us to mine frequent one edges independently, thus eliminating any need for barriers between nodes. Given a data set, all edges which occur at least σ times are used as seeds to recursively partition the search space. These are called *frequent one edges*. Each frequent one edge is grown with all candidate edges. A candidate edge is any other frequent edge, which either grows the graph by adding an additional node, or merely creates a new cycle by adding just an edge. Many previous approaches [3, 11] mined for all frequent edges during each growing step. Recent algorithms [25, 15] only grow graphs with edges that result in graphs known to be in the data set, thus reducing unnecessary work. We incorporate the latter technique. For example, suppose we have a database of graphs $D = \{ABCF, ABDG, ACDG, ABAG\}$ (all these graphs are single paths). Also assume the current graph we are mining is AB. We only attempt to grow AB with candidates C,D and A. We do not attempt to extend AB with B,F and G because ABB, ABF, and ABG did not appear in the data set.

A significant challenge when enumerating frequent subgraphs is that the same graph can be grown from multiple frequent one edges (Figure 1). If both graphs are mined, not only will too many graphs be output, but the time to mine the data set will be significantly longer. One solution is to check the database of previously reported frequent graphs before mining the graph. This is unnecessarily naive, and computationally inefficient. Instead a canonical labeling system is used, which a) uniquely defines a graph, and b) allows us to prune the search space considerably.

3.3 Canonical Labeling

A canonical labeling system is a bijection from graph space to a label space. Several labeling and normalization systems have been proposed [15, 25, 14]. We choose DFScodes by Han [25], because it allows for extensive search space pruning during the graph-growing process, it is universal (all graph types use the same coding), and it is relatively compact. To generate a DFScode for a graph, one traverses the graph in depth first order, and for each edge, appends an edge tuple to the code. An edge tuple is a 5-tuple, namely (start node number, end node number, start node label, edge label, end node label). Node numbers are determined by their order of discovery. Edge tuples which have end node numbers less than their start node numbers are called back edges. Forward edges can only grow from the right-most path. Back edges can only grow from the right-most node. The right-most path of a graph with n nodes is found by starting at node 0, and taking the shortest path to the nth node. The nth node is the right-most node.

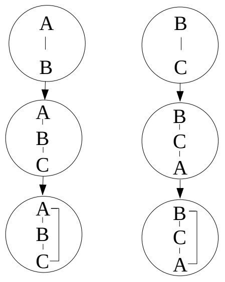


Figure 1: Multiple search paths may produce the same graph.

A graph can have many valid DFScodes, because the DFS traversal can start at an arbitrary node and choose an arbitrary child. To achieve the necessary one to one mapping between a graph and a label, the authors place an ordering on edge tuples. As it is not the contribution of our work, we merely summarize these rules, they are as follow. A DFScode is a string of edge tuples. Two codes are compared at each index of the strings until either a) one edge tuple sorts before the other, or b) one code has no edge at that index (in which case it sorts lower). Back edges sort lower than forward edges. Back edge a sorts before back edge b if the destination of a is lower than that of b. If two back edges have the same node numbers, then we sort by their labels. Forward edge a sorts before forward edge b if the source of a is higher than that of b. If two forward edges have the same start node number, they are sorted by their labels. Interestingly, each valid DFS code for a graph g appears exactly once in a depth first search space traversal. If a code is not the minimal code for a graph, it is not mined, thus avoiding redundant computation. Reconsidering the example in Figure 1, when graph BC was grown with child A, the minimum code check would have failed. For more details on the coding scheme and related correctness proofs, the reader is directed elsewhere [26].

3.4 Candidate Generation

After a DFS code is known to be minimal, it is mined. Because we start with frequent one edges, and only grow frequent children, it is known that the code represents a frequent graph. The database is scanned to generate a list of possible children. Only the graphs which contain the currently mined graph are scanned. For each child occurring in at least σ graphs, we recursively apply the procedure.

3.5 Task Granularity

In our implementation, a task is a pointer based structure with 2 fields. The first field is a DFS code, implemented as a vector of edge tuples. The second is a vector of graph IDs for which the graph appears[2]. We have chosen an algorithm which requires very little state for a task. In addition, mining of each task requires no synchronization between nodes. To process the task, a node merely dequeues the item, and mines the database for the code. We consider two methods to control the granularity of the tasks, which we term partitioning.

3.5.1 Level-wise Partitioning

The goal of level-wise partitioning is to create sufficient tasks to allow the work to be balanced between the nodes. Level-wise partitioning deterministically enqueues tasks to a parameterized depth of recursion. For example, in level-one partitioning, each frequent one edge is enqueued. These tasks are then mined to completion by the dequeuing node. In level-two partitioning, a task is made for each child of a frequent one edge, and then each task is enqueued. Level-two tasks are then mined to completion by the dequeuing node. Level-n partitioning implies level-(n-1) partitioning. For example, level-5 partitioning will enqueue all tasks 5 levels deep in the search space.

3.5.2 Dynamic Partitioning

In dynamic task partitioning, each node makes a decision at runtime for each child task. The task may be mined by the creating node, or enqueued. This decision is based on the current load balancing of the system. We do not require a specific mechanism to make this decision, because in part the decision is based on the queuing mechanism used. For example, with a global queue, it is natural to check the size of the queue. If it is above a minimum value, then the creating node mines the task without queuing; otherwise it enqueues it. With distributed queues, one could set a threshold for the lowest size of any queue. Hierarchical models could set a threshold for any queue in its hierarchy. A more involved decision could be made based on the rate of decay of the size of a queue, a direction for future work.

We believe that if the work is balanced in the system, having the creator of a task mine the child will always be more efficient than enqueing the task, due to the benefits of affinity [13]. The child is always located in a subset of the graphs of the parent, so when those graphs are scanned, the cache will benefit from temporal locality. In addition, there is no queuing costs (although these costs are small in comparison to the size of the smallest task).

[2]We use graph and DFS code interchangeably here

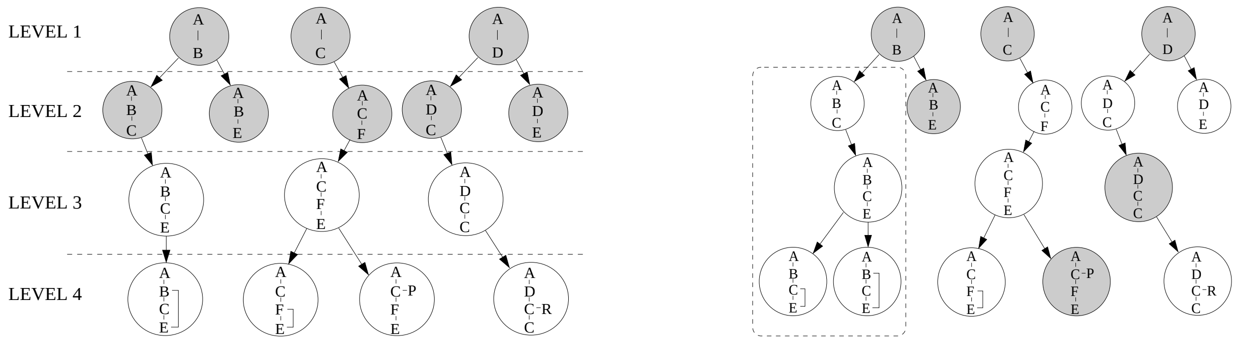


Figure 2: Level-wise Partitioning (left) vs Dynamic Partitioning (right).

Figure 2 depicts these two schemes. In the left figure, level-wise partitioning set to level-2. Independent tasks are shaded gray [3]. In the right figure, a task whose size was allowed to grow dynamically has been circled.

3.6 Task Queuing Model

It is generally accepted that load balancing is the greatest challenge when parallelizing any frequent pattern mining algorithm [4]. To allow a system to allocate tasks to nodes, a queuing model is required. Broadly speaking, two models exist, static task allocation and dynamic task allocation. Under static allocation, tasks are assigned to nodes *a priori*. Because the number of tasks in pattern mining is unknown, one must create an assignment function which maps a task to a node. For example, a hashing function could be used to map DFScodes to a node number, or a simple round robin approach. Then each task, such as mining a frequent one edge, could be assigned to the appropriate processor. These static techniques invariably suffer performance costs. Consider a situation with 5 tasks and 4 processors. If the tasks all require the same mining time, speedup can be at most 2.5 fold, which is 62%. The histograms in Figure 4 illustrate the degree of imbalanced for real world data sets. This problem is understandable. In molecular data sets, for example, the edge C-C is much more frequent than He-C (this is an extreme example since He is a noble gas, however it illustrates the point). With static load balancing, it is conceivable that the two largest jobs in the mining could be assigned to the same node, affording almost no speedup. For this reason, we incorporate a queuing model to accommodate dynamic task allocation. Sev-

SubMine
Input: DFScode s
Output: All σ-frequent graphs.
Method: Call SubMine(s) for each σ-1 DFS s.

```
Procedure SubMine(s)
(1) if (minimumCode(s)) //if s is a canonical label
(2)    AddToResults(s)
(3)    Children C = null
(4)    Enumerate(s,C)
(5)    for each (Child c in C)
(6)        if (c is frequent)
(7)            DFScode newS = Append(c,s)
(8)            if (timeToMine)
(9)                SubMine(newS)
(10)           else
(11)               Enqueue(newS)
```

Figure 3: Pseudo Code

[3]As an optimization, each node actually keeps 1 child task, and enqueues the others.

eral queuing models are available; we describe them below. Each can be either First-In-First-Out (FIFO) or Last-In-First-Out (LIFO).

3.6.1 Global Queue

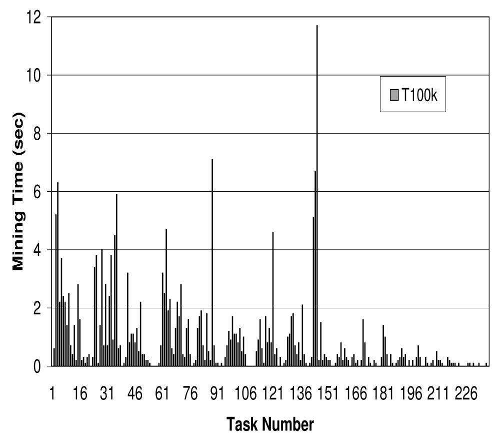
A global (or central) queuing model is a model in which each node adds and removes tasks from the same queue. A single mutex is required for all queuing and dequeuing. Contention is an obvious concern, because operations on the queue are serialized. One benefit is that there is maximum task sharing amongst nodes. Any task enqueued into the system is readily available to any idle node. Another benefit is ease of implementation. If a node finishes a task, and there are no other tasks in the global queue, it sleeps. When a node enqueues a task, it awakens all sleeping nodes. Termination occurs when a node finds no tasks in the queue and all other nodes are sleeping.

3.6.2 Hierarchical Queues

Hierarchical queuing is designed to allow task sharing between nodes. Each node has its own dedicated queue, which does not have a mutex. Nodes enqueue and dequeue from their own queue. If this queue is full, a node will enqueue into a shared queue. Conceptually, this queue is a level above the dedicated queues. This shared queue can either be above a subgroup of nodes, or over all nodes (i.e. easily extendable to more than two levels). If it is over a subset, then only that subset adds and removes from it. When a node's dedicated queue is empty, it attempts to dequeue from the shared queue. This shared queue has a mutex, and occurs some contention. If the shared queue is for a subset of nodes, then there will be another shared queue above it, which is for multiple node groups. This hierarchical structure ends with a queue with unlimited capacity at the top, which is shared by all queues. A node sleeps when its queue and all queues above it are empty. If a node spills a task from a local queue to a global queue, it awakens the other nodes in the group. Termination occurs when a node finds no other tasks, and all other nodes are sleeping. The hierarchical model allows for task sharing, while affording fast enqueue and dequeue most of the time because the local queue has no lock.

3.6.3 Distributed Queues

In the distributed queuing model, there are exactly as many queues as nodes. Each queue is unlimited in capacity, has a mutex, and is assigned to a particular node. The default behavior for a node is to enqueue and dequeue using its own queue. Although this incurs a locking cost, there is generally no contention. If a node's queue is empty, it searches other queues in a round robin fashion, looking for work. If all queues are empty, it sleeps. When a node enqueues a task, it wakes all sleeping nodes. Before a node sleeps, it checks whether other nodes are sleeping. If all other nodes are sleeping, the algorithm terminates. The distributed model affords more task sharing than the hierarchical model, at the cost of increased locking. Also, as the size of the local queue decreases in the hierarchical model, it more closely resembles the distributed model.



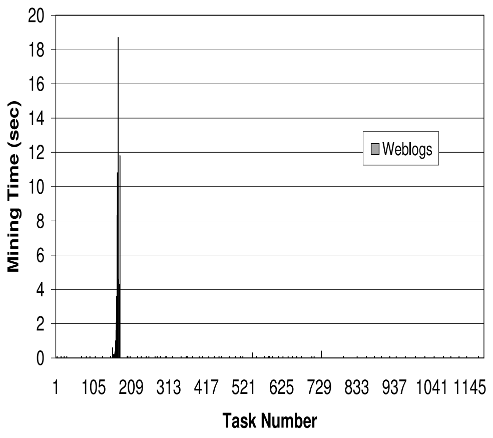


Figure 4: Task time histograms for level-3 partitioning with synthetic data (left) and real data (right).

3.7 Embedding Lists

An embedding list is a pointer based structure containing a mapping between the current mined graph and its location in the database. There may be multiple mappings to a particular database graph, because the current graph may appear in it more than once. Embedding lists have been shown to provide a performance improvement [15] in serial graph mining implementations because they allow for fast discovery of child graph candidates. However, the dependence on the memory structure is costly in a parallel setting, because all child graphs of the same parent must be mined before the parent's state can be freed. This effectively creates a synchronization constraint, and stresses the memory subsystem. For these reasons, we do not employ embedding lists. It has been shown by [24] that efficient depth first algorithms without embedding lists can outperform many algorithms with embedding lists. This is a benefit of gSpan, as it does not require embedding lists to mine graphs efficiently.

3.8 List Joins

Each graph has a list of potential candidates in which it can be grown. A true depth-first traversal of the search space would grow the first candidate, and mine it to completion before mining the next candidate. Recently an alternate method has been adopted, which provides a potential improvement to execution times. Each candidate is evaluated for its frequency and graph occurrences. Then one candidate is mined. In the next recursive mining of this candidate, it reviews its siblings. If a sibling can provide insight into the potential frequency of one of its candidates, then this information is used. The two lists are union-ed, forming the child's graph occurrence list. Consider a straight graph ABC which has three child, namely ABCD, ABCE and ABCF. When child ABCD is mined, it is guaranteed to generate child candidate ABC(DE) since ADCE is frequent. Notice that ABC(DE)'s graph occurrence list is the union of the lists from its parent, ABCD, and one of its parent's siblings, ABCE. A quick list join operation produces the graph occurrence list.

Although helpful in a serial implementation, it is restrictive in a parallel setting, because a) the memory costs for maintaining the parent's sibling data are higher, and b) the data dependency generated between the tasks. After an empirical evaluation, we removed list joins from our algorithm to improve load balancing, and this is the implementation we evaluate in the next section.

The basic algorithm is detailed in Figure 3. In line 1 the DFScode s is verified to be minimal. In line 4 all the graphs which contain the graph represented by s are mined for potential candidates. In lines 5-11, all candidates which occur in at least σ graphs are recursively mined. Line 8 checks a boolean condition, which is based on the partitioning scheme and granularity used, to decide whether to mine the child directly, or enqueue the task.

4 Experimental Results

We implement our algorithm in C using POSIX threads. All experiments allocate one thread per processor, which we term a node. With one processor, our algorithm is comparable in execution time to the original gSpan binary, kindly provided by the authors. This can be seen in Table 5. Unfortunately, we cannot compare the codes on data sets with more than 256 node labels, as that was the maximum allowable for their binary. All experiments are run on an SGI Altix 3000 with 64 GB of memory, 32 1.3 Gigahertz Intel Itanium 2 processors, 4 Gigabit Ethernet interfaces, 2 Gigabit Fiber Channel interfaces, and approximately 400 GB of temporary disk space, unless otherwise noted.

4.1 Data Sets

We employ several real world data sets, shown in Table 1. PTE1 is a data set of molecules classified as carcinogens by the Predictive-Toxicology Evaluation project [4]. HIV1 is a data set of molecules which have been screened for anti-HIV1 activity by the National Cancer Institute [5]. Weblogs is a data set of web sessions, generated from web server request and response logs[19]. T100k is a synthetic data set made from the PAFI toolkit[6].

	PTE1	HIV1	D10k
gSpan (authors)	198 sec	322 sec	36 sec
our code	204 sec	270 sec	33 sec

Figure 5: A comparison of single node execution times.

4.2 Scalability

We measure both weak and strong scalability. To measure strong scalability, we used the full data set and increased the number of nodes, and set support such that total mining times between data sets were comparable. On 32 nodes, the scalability ranges from 27.4 to 22.5 over the four data sets, as seen in Figures 7 and 6. Weblogs has the lowest scalability at 22.5. This can be attributed

[4]http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/PTE/
[5]http://dtp.nci.nih.gov/docs/aids/aids_data.html
[6]http://www-users.cs.umn.edu/ karypis/pafi/

Data set	# Graphs	# Node labels	# Edge labels
Weblogs	31181	9061	1
PTE1	340	66	4
HIV1	422	21	4
T100k	100000	1000	50

Table 1: Data sets used for experiments.

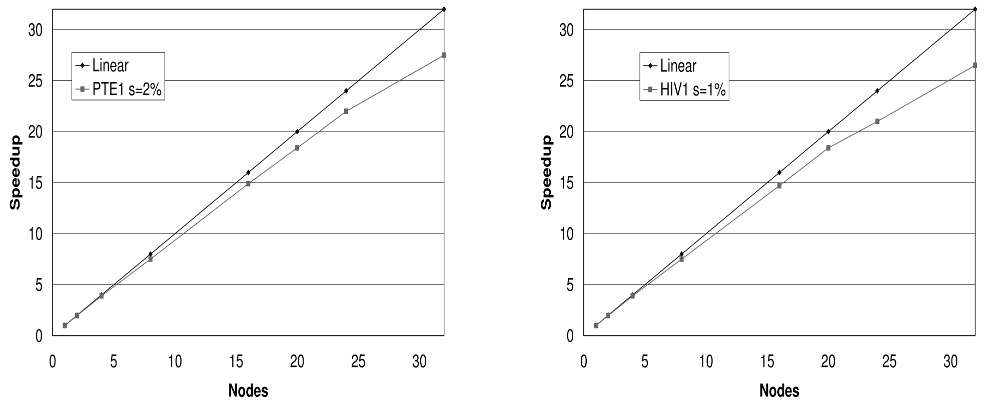


Figure 6: Strong scalability of the PTE1 data set (left) and HIV1 data set (right).

to its increased memory traffic. The frequent graphs in the Weblogs data set are larger than those in the other data sets, which requires the algorithm to traverse a larger percentage of each of the database graphs, deeper into the recursion. Scalability generally increases as support is reduced, because there are more tasks, and those tasks can be mined by the creating node.

To measure weak scalability, we partitioned the synthetic data set T100k into 32 equal chunks, and executed the algorithm with a number of chunks proportional to the number of nodes. Ideally, we would scale work precisely with increasing nodes. Graph mining runtimes are highly dependent on the associativity and distribution of the data, so we tuned support in each experiment to return the a graph result set proportional to the number of nodes. We use the synthetic data set T100k for this experiment because we believe its partitions more closely resembles the properties of the full data set. The results are presented in Figure 8. The algorithm accommodates the increase in data set size without a degradation in performance.

4.3 Queuing Model Evaluation

To evaluate the queuing models, we ran all three models on several data sets. We present the results of Weblogs here, in Figure 8. One global queue was not significantly worse than distributed queues. Contention on the global queue is not a major limiting factor in scalability at 32 nodes. Still, dynamic queues provide the best performance of the three. Hierarchical queues perform poorly, because task sharing is too low. As seen in Figure 4, the Weblogs data set has only a few frequent one edges with a significant number of child jobs. These jobs are not propagated to distant

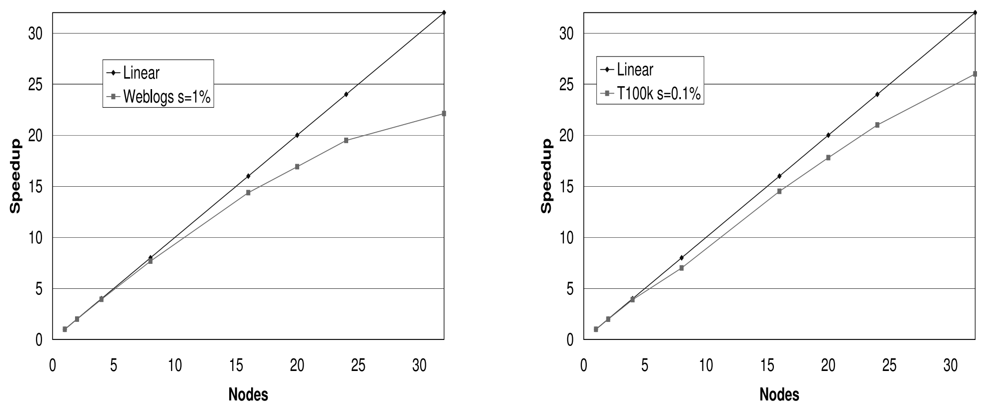


Figure 7: Strong scalability of the Weblogs (left) and T100k (right) data sets.

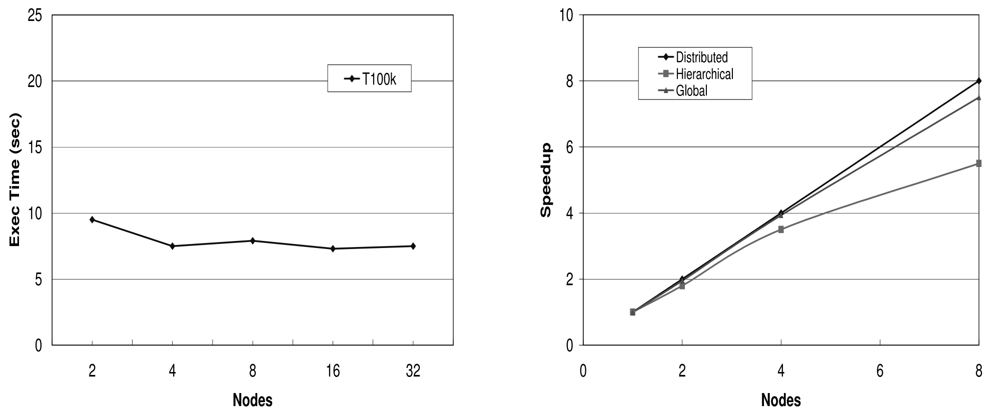


Figure 8: Weak scalability results for T100k (left), and queuing model scalabilities (right).

nodes in the hierarchical queue structure.

FIFO vs LIFO had no statistically significant impact. This is because once a node queues a task, it is unlikely to have high temporal locality when it is dequeued, regardless of where it is located in the queue (top or bottom). Some data sets with low frequent one tasks benefit slightly with a FIFO queue because the children of the large tasks are more likely to be large as well, and the sooner they are distributed, the better the load balancing. Other data sets benefit slightly from LIFO because the most recently queued jobs have better temporal locality if they are dequeued within a few recursions from their queuing time.

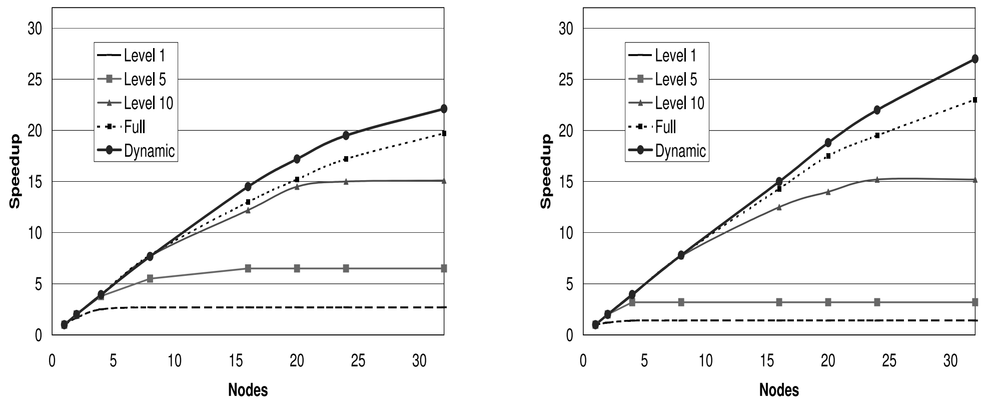


Figure 9: Partition Model Scalabilities on Weblogs (left), and HIV1 (right).

4.4 Partitioning Model Evaluation

To evaluate our task partitioning strategies, we employ the Weblogs and HIV1 data sets. Both data sets exhibit a high differential for frequent one edge mining times, and are difficult to properly load balance. We use distributed queues for these experiments. The results are presented in Figure 9.

Level-wise partitioning of the search space does not provide an even distribution of the tasks. Even partitioning to 10 levels into the recursion tree, scalability is hindered. The differences in scalability between these partitioning models is primarily due to load balancing. Full partitioning enqueues each child task, regardless of the state of the system. It outperforms all level-wise partitioning depths (1,5,10, and others not shown here). The performance difference between full partitioning and dynamic partitioning is due to the poor cache performance exhibited in full partitioning.

We perform a working set study to compare full partitioning and dynamic partitioning. We use cachegrind [7] on a single processor machine (Pentium 4 2GHz with 1GB RAM). Because Cachegrind currently does not profile multiple threads, we simulate 32 threads by allowing a single thread to remove from any point within 32 locations from the tail of the queue with equal probability. As seen from the results in Figure 10, temporal locality deteriorates when enqueuing all tasks. Each node keeps at least one child in all models, but in full partitioning, all other tasks have an increased probability to be mined not directly after their parent task.

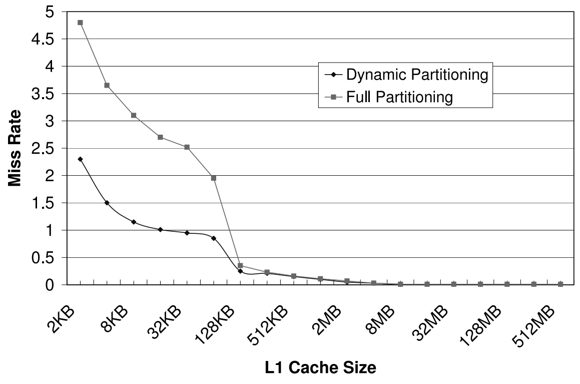


Figure 10: Working Sets for Full-Partitioning vs Dynamic Partitioning.

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

5 Discussion and Future Work

Throughout the project, our focus has been on removing the bottlenecks of scalability. The first bottleneck was load balancing, which we addressesd with dynamic partitioning and an appropriate queuing strategy. Once we could guarantee all the nodes were busy thoughout the execution, we found that parallel calls to malloc() were a problem. In a shared environment, heap allocation and deallocation are handled serially, so as to avoid returning the same memory address to two concurrent requests. Most operation in graph mining use data containers without a predetermined size, so they use heap space. For example, when searching for a graph in the database, it is not known apriori in how many graphs it will be embedded. This is not an uncommon problem in shared memory environments, and researchers have proposed solutions [7]. Our solution was to give each node its own stack of heap space for recursive calls. Because the algorithm is a depth-first recursion, markers can be set for each level of recursion, so as to allow simple deallocation without a loss of usable memory.

The current bottleneck to improved scalability is high memory traffic. In the worst case, bus utilization overwhelms the system, increasing from 0.2% on 1 node to 25.8% on 16 nodes (details omitted due to space constraints). We are in the preliminary stages of research designed to limit memory access by employing dynamic state management. We believe that multi core systems will have limited resources for off-chip bandwidth, and that shared memory algorithms will need to monitor memory activity closely. Mutli core systems have advantages over conventional smp architectures such as reduced communication costs and the potential for shared L2 caches. Our own preliminary studies using a custom simulator suggest that these differences can be leveraged with dynamic state management, where recursive mining calls between parent and child tasks can incorporate more state than enqueued tasks without additional overhead. A node who mines its own child can use in-cache reduced embedding lists to lessen the subgraph isomorphism test inherit in the algorithm.

6 Conclusion

We have shown that an algorithm which allows the state of the system to control the behavior of task allocation and granularity can greatly improve parallel graph mining performance. In addition, we have illustrated that distributed queuing provides the best mix of sharing and locking for this workload. Through these techniques, we have developed a parallel frequent graph mining algorithm which can decrease mining runtimes by up to a factor of 27 on 32 nodes. In the near future, we plan to improve upon this by incorporating dynamic state management for shared cache architectures such as future multi core systems.

References

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 1993.

[2] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 1995.

[3] C. Borgelt and M. R. Berthold. Mining molecular fragments: Finding relevant substructures in molecules. In *Proceedings of the 2nd Internation Conference on Data Mining (ICDM)*, pages 51–58, 2002.

[4] S. Cong, J. Han, J. Hoeflinger, and D. Padua. A sampling-based framework for parallel data mining. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 255–265. ACM Press, 2005.

[5] D. J. Cook, L. B. Holder, G. Galal, and R. Maglothin. Approaches to parallel graph-based knowledge discovery. volume 61, pages 427–446, Orlando, FL, USA, 2001. Academic Press, Inc.

[6] L. Dehaspe, H. Toivonen, and R. D. King. Finding frequent substructures in chemical compounds. In *Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining*, pages 30–36. AAAI Press, 1998.

[7] R. D. B. Emery D. Berger, Kathryn S. McKinley and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *The Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, 2000.

[8] V. Guralnik and G. Karypis. Dynamic load balancing algorithms for sequence mining. In *Univeristy of Minnesota Technical Report TR 00-056*, 2001.

[9] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2000.

[10] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *Proceedings of the 3rd Internation Conference on Data Mining (ICDM)*, pages 549–552. IEEE Press, 2003.

[11] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proceedings of the Internation Conference on Data Mining (ICDM)*, pages 313–320, 2001.

[12] H. Manilla, H. Toivonen, and I. Verkamo. Discovering frequent episodes in sequences. In *Proceedings of the International Conference on Knowledge Discovery and Data mining (KDD)*, pages 146–151, 1995.

[13] E. Markatos and T. Leblanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. In *IEEE Transactions on Parallel and Distriuted Systems*, 1993.

[14] B. McKay. Practical graph isomorphism. In *Congressus Numerantium*, volume 30, pages 45–87, 1981.

[15] S. Nijssen and J. N. Kok. A quickstart in frequent structure mining can make a difference. In *Proceedings of the 10th International Conference on Knowledge Discovery and Data mining (KDD)*, pages 647–652, New York, NY, USA, 2004. ACM Press.

[16] S. Parthasarathy. Active data mining in a distributed setting. In *University of Rochester Ph.D. Thesis*, 2000.

[17] S. Parthasarathy and M. Coatney. Efficient discovery of common substructures in macromolecules. In *Proceedings of the IEEE International Conference on Data Mining (ICDM)*, 2002.

[18] S. Parthasarathy, M. J. Zaki, M. Ogihara, and W. Li. Parallel data mining for association rules on shared-memory systems. In *Knowledge and Information Systems*, volume 3, pages 1–29, 2001.

[19] J. Punin, M. Krishnamoorthy, and M. J. Zaki. Logml – log markup language for web usage mining. In *WEBKDD Workshop: Mining Log Data Across All Customer TouchPoints (with SIGKDD01)*, 2001.

[20] P. Raghavan. Social networks on the web and in the enterprise. In *Lecture Notes in Computer Science*, volume 2198, 2001.

[21] A. Srinivasan, R. King, S. Muggleton, and M. Sternberg. Carcinogenesis predictions using ilp. In *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297, pages 273–287. Springer-Verlag, 1997.

[22] A. Termier, M.-C. Rousset, and M. Sebag. Treefinder: a first step towards xml data mining. In *International Conference on Data Mining ICDM02*, 2002.

[23] C. Wang and S. Parthasarathy. Parallel algorithms for mining frequent structural motifs in scientific data. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*, 2004.

[24] M. Wrlein, T. Meinl, I. Fischer, and M. Philippsen. A quantitative comparison of the subgraph miners mofa, gspan, ffsm, and gaston. In *The 9th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD)*, Porto, Portugal, 2005.

[25] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *Proceedings of the 2002 International Conference on Data Mining (ICDM)*, 2002.

[26] X. Yan and J. Han. gspan: Graph-based substructure pattern mining, expanded version. In *UIUC Technical Report*, volume UIUCDCS-R-2002-2296, 2002.

[27] X. Yan, X. J. Zhou, and J. Han. Mining closed relational graphs with connectivity constraints. In *Proceedings of the 11th International Conference on Knowledge Discovery and Data mining (KDD)*, 2005.

[28] M. Zaki. Parallel sequence mining on shared-memory machines. In *Journal of Parallel and Distributed Computing*, volume 61, pages 401–426, 2001.

[29] M. J. Zaki. Parallel and distributed association mining: A survey. In *IEEE Concurrency, special issue on Parallel Mechanisms for Data Mining*, volume 7.4, pages 14–25, 1999.

[30] M. J. Zaki, V. Nadimpally, D. Bardhan, and C. Bystroff. Predicting protein folding pathways. *Bioinformatics*, 20(1):386–393, 2004.