Towards Data Mining on Emerging 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

Recent advances in microprocessor design have given rise to new commodity architectures. One such innovation is to place multiple cores on a single chip, called Chip Multiprocessing (CMP). Each core is an independent computational unit, allowing multiple processes to execute concurrently. A second recent architectural advancement is to allow multiple processes to compete for resources simultaneously on a single core, called simultaneous multithreading (SMT). SMT can improve overall throughput in cases where CPU utilization is low. We investigate the implications of these advances on the design of data mining algorithms. In particular, we focus on frequent graph mining. Mining graph based data sets has practical applications in many areas including molecular substructure discovery, web link analysis, fraud detection, and social network analysis. In this work, we propose a novel approach for parallelizing graph mining on CMP architectures. We design a parallel algorithm with low memory consumption, low bandwidth, and fine task granularity. We show that dynamic partitioning and dynamic task allocation provide a synergy which greatly improves scalability over a naive algorithm, from 5 fold to 27 fold on 32 nodes.

1 Introduction

Recent advances in microprocessor design have transformed the desktop PC into a shared-memory parallel machine. Two such advances are chip multiprocessing (CMP) and simultaneous multithreading (SMT). CMP designs exhibit multiple processing cores on one chip. These cores are independent computational units which may or may not share level 2 cache. Current dual core chips include Intel's Pentium D, AMD's Opteron, and IBM's Power4.

SMT provides support for multiple contexts to compete simultaneously for chip resources, in an effort to improve overall utilization. It is essentially a hardware-based latency-hiding technique. Intel's Hyperthreading technology is an example of SMT. By executing two threads concurrently, an application can gain significant improvements in effective IPC (instructions per cycle).

These technologies arise in part because of the inherent challenges with increasing clock frequencies. The increase in processor frequencies over the past several years has required a significant increase in voltage, which has increased power consumption and heat dissipation of the central processing unit. In addition, increased frequencies require considerable extensions to instruction pipeline depths. Finally, since memory latency has not decreased proportionally to the increase in clock frequency, higher frequencies are often not beneficial due to poor memory performance. By incorporating thread level parallelism, chip vendors can continue to improve IPC by exploiting thread level parallelism, without raising frequencies. As these low cost parallel chips become mainstream, designing data mining algorithms to leverage them becomes an important task. Finally, it is likely that even distributed clusters will groups of inexpensive, highly parallel shared memory machines.

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 [20], XML documents [19], web link data [16], financial transaction data [9], social network data[17], protein folding data [24], 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 convoluted or inefficient when captured in relational tables. For example, molecular data is easily represented in graphical form. Also, financial transactions can be stored as graphs, where each node is an institution and the edge

*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).

is an account number. Finally, web logs can be mapped to user sessions, and subsequently represented as graphs [16].

Frequent pattern mining was first introduced by Agrawal et al [1] in 1994, and has been studied extensively [5, 8, 13, 20, 21]. In web session data, frequent patterns represent resources which are more often used, and the paths most often taken to access those resources. Application service providers can improve web server behavior by prefetching resources as users move through 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, researchers exploring immunodeficiency viruses [7] may gain insight into recombinants via commonalities within initial replicants. In addition, it can be used to group previously unknown groups, affording new information. Zaki et al [24] apply graph-based methods on weighted secondary graph structures of proteins to predict folding sequences. Finally, in financial transaction data, large graphs with high occurrence rates could represent potential criminal activity such as money laundering.

Although shared memory algorithms exist for several data mining tasks, none of these works address the architectural issues specific to CMP systems we outline in this work. The key contributions of this paper are

- We describe several key issues to consider when designing data mining algorithms for CMP machines
- We provide a highly scalable parallel graph mining algorithm, specifically designed to maximize scalability on CMP architectures
- We design several task partitioning models, and show that a novel dynamic model affords the lowest runtimes

2 Related Work

Initial efforts exist [18, 6] which explore data mining on clusters of SMP machines. Jin and Agrawal [6] developed a parallel algorithm for association rule mining on a cluster of SMP machines. They effectively partitioned the workload to make use of multiple CPUs on a node. This was extended to dense datasets by Sucahyo et al [18].

Several serial algorithms exist which enumerate frequent subgraphs. FSG [8] mines for all frequent subgraphs in an APRIORI-like breadth-first fashion. Yan and Han proposed GSpan [21], which uses a combination of depth-first and breadth-first trajectories to discover frequent patterns. Gaston [13], developed by Nijssen and Kok, is an efficient 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. Meinl et al [12] parallelized MoFa[2] for SMP machines, and kindly supplied their implementation. We will briefly compare our parallel algorithm to it in Section 4.

Wang and Parthasarathy [20] developed a parallel algorithm for their Motif Miner toolkit [14]. Their parallelization strategy cannot be directly applied to the more general graph mining problem. Cook et al [3] have been developing a system called Subdue for several years. Their research has produced a parallel implementation, which solves a 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 [15, 23]. Strategies for mining associations in parallel have been proposed by various researchers. Zaki [22] proposed parallel partitioning schemes for sequence mining. The author illustrates the benefits of dynamic load balancing in shared memory environments.

None of these works evaluate data mining techniques on new architectures, such as CMPs. In a previous effort [4], we examined frequent itemset mining on SMT architectures.

3 Graph Mining on Emerging Architectures

3.1 Architectures

Chip Multiprocessing (CMP) designs (often called multicore) incorporate more than one processing element on the same die. They can execute multiple threads concurrently because each processing element has its own context and functional units. Most designs will run these elements at somewhat slower clock frequencies than current state of the art, to conserve power. Although current designs have relatively few cores, Intel's 2015 Processor road map proposes CMPs with hundreds of cores [1]. Multicore serves both basic desktop needs as well as the needs of the high performance community. Desktop users gain true multitasking for everyday tasks such as multimedia and office applications. High performance computing will gain considerable parallelism opportunities, as these chips become standards for cluster nodes.

CMP systems have significant differences when compared to existing platforms. To begin with, *inter-process communication costs can be much lower than previous*

[1] http://www.intel.com/technology/magazine/computing/platform-2015-0305. htm

parallel architectures. Distributed clusters must exchange messages, and traditional shared memory multiple CPU systems write to main memory. CMP systems can keep locks on chip, either in specially designed hardware, or in a shared L2 cache. A direct consequence of lower locking costs is an improved task queuing system. Because CMP chips are designed for thread level parallelism, low-overhead software queues are essential; on chip hardware-based queues are also attractive. Therefore, *algorithm designers should consider a much finer granularity* when targeting these systems. For example, in FPGrowth[5], multiple threads could mine items from the same projected header table.

Several important issues with CMP systems arise when discussing the memory hierarchy. To begin with, *off chip bandwidth is a precious commodity*. Whereas clusters have bandwidth which scales with increasing processing elements, CMP systems do not. CMPs are even more limited than SMPs in this regard, because each node in an SMP has independent access to the memory subsystem. CMP systems are limited because of the practical limits to the number of pins which can be placed on a single chip. Although not significant with two cores, as chips scale to 32 and 64 cores off chip bandwidth will be a significant concern. Algorithm designers can compensate by improving thread-level data sharing.

Intuitively, CMP architectures increase the potential computational throughput while maintaining the same amount of main memory as their single core counterparts. The amount of main memory in commodity CMP systems will likely not rival that of high end SMP or distributed systems. *We believe that algorithms which trade slightly increased computation for lower memory consumption will exhibit better overall performance on CMP systems*, when compared to systems which use more main memory.

CMP systems can have shared caches. Although both AMD and Intel currently manufacture dual core chips with dedicated L2 caches, future designs possess shared L2 cache. Essentially, having a shared L2 cache allows the on chip memory of each processing element to grow based on runtime requirements. Cache coherency protocols can be processed on chip, *allowing multiple processing elements to share a single cache line and save a significant number of memory operations.* For example, the Intel Duo's Smart Cache does not need to write to main memory when multiple cores write to the same cache line. This advancement greatly improves the potential benefits of cooperation amongst processes.

Simultaneous multithreading (SMT) is the ability of a single processing element to execute instructions from multiple independent instruction streams concurrently. Because SMT systems do not have additional functional units, it's main performance improvement is to hide the latency of main memory when executing multiple threads. Many data mining workloads are latency bound, and as such can benefit from SMT. *Algorithm designers should intelligently schedule thread execution to share as many memory fetches as possible*, a process called thread co-scheduling.

3.2 Frequent Graph Mining

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.* In our instantiation of the problem, edges and vertices have labels. If the dataset does not contain labels, we merely set all labels to 0. Edges can be either directed or undirected. In building an algorithm for CMP, we seek to establish a small memory footprint, low bandwidth utilization, and fine-grained task partitioning. In addition, we propose to incorporate as much runtime information as possible in an effort to maximize system resource utilization.

To reduce the exponential time complexity required to test for subgraph isomorphism, a canonical labeling system is used. A canonical labeling system is a bijection from graph space to a label space. Several labeling and normalization systems have been proposed [13, 21, 11]. Our labeling closely resembles DFScodes by Han [21].

Given a data set, all edges which occur at least σ times are called *frequent one 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. Graph mining algorithms can traverse the search space in either a depth-first fashion, or a breadth first fashion. Each frequent one edge is grown with all possible candidate edges. This process then recurses to mine for all possible frequent subgraphs in the dataset.

3.3 Algorithmic Improvements

We now provide our techniques for addressing the CMP specific issues outlined in section 3.1 for graph mining.

3.3.1 Improving Task Granularity

In our algorithm, every child candidate can be a separate task. We consider two methods to control the granularity of the tasks, which we term partitioning. Level-wise partitioning deterministically enqueues tasks to a parameterized depth of recursion. Dynamic partitioning allows the parent mining node to decide at runtime whether to enqueue each child task, or to mine it locally. This decision is based on the current load balance 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, an involved decision could be made based on the rate of decay of the size of the appropriate queue, perhaps implemented with on chip hardware. Dynamic partitioning affords very fine-grained task allocation, without the queuing costs associated with full partitioning.

We use a distributed queuing model, where there are exactly as many queues as processing nodes. Each queue is unlimited in capacity, has a lock, 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; an enqueing thread then wakes sleeping threads. We believe that distributed queues are the best fit for CMP architectures because their round robin checking mechanism will be relatively inexpensive, while affording highly efficient dynamic load balancing.

3.3.2 Minimizing Off-chip Bandwidth

Our parallel algorithm employs a depth-first search space traversal. We only grow graphs with candidates that are found to be frequent in the projected dataset. As stated prior, each frequent substructure is grown with all observed candidates. Briefly, let g be a frequent graph, and S be the set of graphs containing g. The locations of g in S are called g's embeddings. Let g' be the result of appending a candidate α to g, and let S' be its graph set. It is clear that $S' \subseteq S$. To increase temporal locality, and lower off-chip bandwidth, we maximize the possibility g' is processed by the miner of g. We accomplish this through our distributed queuing, and placing a bound on the total number of tasks that exist in all the queues. When g' is mined, the cache will benefit from temporal locality of mining g. Performance is increased via the benefits of affinity [10].

3.3.3 Lowering Memory Consumption

By limiting the amount of main memory consumed, we increase the size of the problem we can solve. Returning to our definitions above, each embedding of g' overlaps a corresponding embedding of g in S', and thus maintaining g's embedding list will aid in constructing the embeddings for g'. There is a performance trade off when considering the amount of memory used to maintain embeddings. In our algorithm, a task is a pointer-based structure with 2 fields. The first field is the code for the graph, implemented as an array of tuples. The second is a vector of pointers to the first node of each embedding of g''s parent. Thus, we maintain very little state when queuing a task. We do not employ expensive embedding lists[13], nor do we utilize list joins[21]. List joins require that every child's extensions be known, which requires more main memory because all g' for a given g must be in memory concurrently. Our succinct embedding list affords two benefits, namely a) low memory consumption and b) there is no data dependency between tasks.

```
SubMine
Input: Code s
Output: All frequent graphs.
Method: Call SubMine(s) for each σ-1 code s.
Procedure SubMine(s)
(1) if (minimumCode(s))
(2)   AddToResults(s)
(3)   Children C = null
(4)   Enumerate(s,C)
(5)   for each (Child c in C)
(6)     if (c is frequent)
(7)       Code newS = Append(c,s)
(8)       if (timeToMine)  SubMine(newS)
(10)       else  Enqueue(newS)
```

Figure 1: Pseudo Code

3.3.4 Improving Thread Cooperation for SMT

To improve efficiency on SMT systems, we force two threads to share the same queue, and increase the bound before the queue is considered full. They are initialize with a parent and child of the same frequent one item. The net effect is to create a system where two threads are mining tasks in the same portion of the search space, thus they share a large percentage of the database graphs residing in cache.

4 Experimental Results

4.1 Experimental Setup

We implemented our algorithm in C++ using POSIX threads. Experimentation was performed on several real machines. We used a 2.8 GHz dual core Pentium D with 2 GB of RAM, a 2.80GHz Intel Hyperthreaded Pentium 4 with 1GB of RAM, and an Altix 3000 with 32 1.3 MHz processors and 64 GB of RAM. We employ several real world data sets, shown in Table 2. Briefly, these include PTE1 [2], HIV1 [3], the nasa multicast dataset[4], and Weblogs

[2] http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/PTE/
[3] http://dtp.nci.nih.gov/docs/aids/aids_data.html
[4] http://www.nmsl.cs.ucsb.edu/mwalk/

[16]. T100k and T50k are synthetic data sets made from the PAFI toolkit[5].

4.2 Scalability

Our algorithm compares favorably to other public graph miners. GSpan[21] and MoFa[12] were both kindly supplied by their original authors. The provided GSpan is serial and requires there to be no more than 256 node labels (shown as –). Parallel MoFa is implemented in Java, which may hinder its overall performance. Noting this, our CMP algorithm scales considerably better both in terms of memory consumption and execution times. A key factor is that our algorithm does not require that an additional thread consume additional memory. Execution times for several datasets executed on the dual core are given in Table 1. All datasets are processed in nearly half the time, except the nasa dataset. Interestingly, while only scaling 1.62 times on the dual core machine, it scales 1.95 fold on the Altix. After inspection, we found that this dataset has a much larger working set due to its highly repetitive large tree patterns. Each node on the Altix is equipped with a 256KB L2 cache and a 3MB L3 cache, whereas the Pentium D 820 has merely a 1MB L2 cache, and no L3 cache. This additional cache made a considerable difference in this case.

On the 32 node Altix, the scalability ranges from 21.5 to 27.4 over the four data sets, as seen in Figure 2. This is a major improvement over existing techniques[12], which scale to 7 fold on 12 nodes. Weblogs has the lowest scalability. This can be attributed 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.

Our algorithm uses much less memory than other parallel shared memory solutions, as presented at the right of Figure 2. This study was performed on the PTE dataset at 1% support. As illustrated, our CMP algorithm for graph mining does not consume significantly more memory as the number of threads increases (increase is about 10%). This is intuitive, since threads do not maintain state, and the working memory required for any one thread is typically much lower than the dataset. Finally, scalability generally increases as support is reduced, since the common case (in dynamic partitioning) is for the parent to mine its own children (thus better locality).

Full partitioning enqueues each child task, regardless of the state of the system. It outperforms all level-wise partitioning depths (1,5,10, and other static methods, not shown here). The right graph in Figure 2 illustrates the benefits of dynamic partitioning on the PTE dataset. On average, we found that partitioning based on frequent one items resulted in poor load balancing, affording merely a 5 fold speedup on 32 nodes. Fine-grained full partitioning more closely approached dynamic partitioning. The performance difference between full partitioning and dynamic partitioning is due to the slightly lower cache performance exhibited in full partitioning. A detailed simulation is currently in progress to study this further.

4.3 Simultaneous Multithreading

We briefly evaluate the benefit of SMT using the hyperthreaded machine described in Section 4.1. As seen in Figure 3, most datasets see only a modest improvement when compared to dual core execution time reductions. This is because the CPU utilization is high. However, as the working set size increases, hyperthreading has significant benefits. The nasa dataset illustrates this notion. We performed a cache working set study to simulate the difference between co-scheduled threads and running independent threads. We simulated this experiment using cachegrind[6], allowing one thread to alternate between independent tasks in case 1, and mining from similar parent tasks in case 2 (co-scheduling). Figure 3 illustrates the increased miss rates of two independent threads. This same effect occurs if we partition the dataset completely as opposed to allowing parent threads to mine as many of their children as possible. In future work we will implement a tightly coupled co-scheduling algorithm to evaluate this further.

5 Conclusions and Future Work

Several architectural advancements have led to the emergence of new inexpensive commodity parallel systems. These systems have significant differences from existing architectures, and pose new challenges to algorithm designers. We have illustrated the benefits of mindful algorithm design with a sample data mining task, frequent substructure mining. We feel this algorithm is well suited for commodity next generation systems because of its low memory bandwidth, small footprint, fine granularity, and frequent communication (via queues). In the near future, we plan to improve upon this by incorporating dynamic state management for shared cache architectures. We also plan to investigate the scalability of large CMP systems via a publicly available simulator. Finally, we plan to investigate tight co-scheduling to further evaluate the benefits of SMT.

[5] http://www-users.cs.umn.edu/ karypis/pafi/

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

	PTE1 2%		HIV1 2%		Weblogs .08%		nasa 85%		T50k .01%	
	1	2	1	2	1	2	1	2	1	2
our CMP alg (C++)	69	36	42	23	64	37	99	61	140	74
MoFa (authors Java)	5901	3498	615	333	format not supported		115	70	2868	out of mem
gSpan (authors C++)	79	–	35	–	–	–	–	–	–	–

Table 1: Execution times (in seconds) for 1 and 2 threads on a dual core Pentium D processor.

Data set	# Graphs	# Node labels	# Edge labels
PTE1	340	66	4
HIV1	422	21	4
Weblogs	31181	9061	1
nasa	1000	342	1
T50k	50000	1000	10
T100k	100000	1000	50

Table 2: Data sets used for experiments.

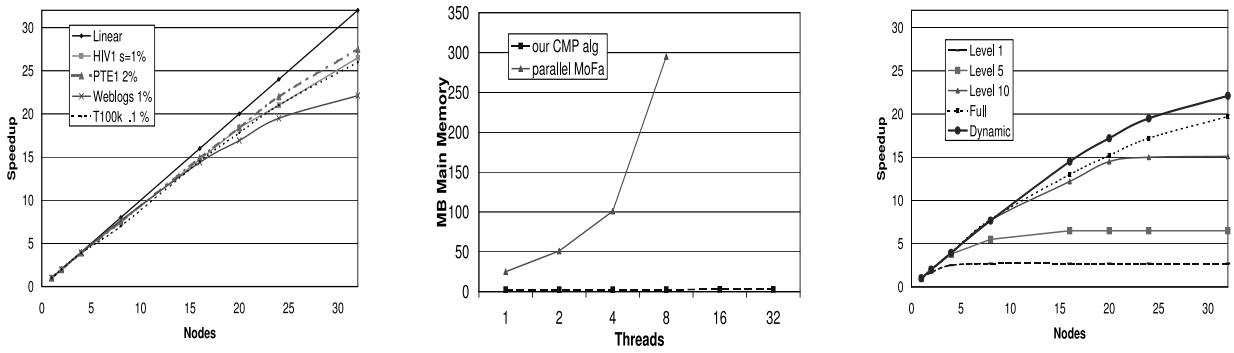

Figure 2: Scalability for several datasets on an Altix 3000 (left), memory consumption with increasing threads (middle), and dynamic partitioning vs static partitioning (right).

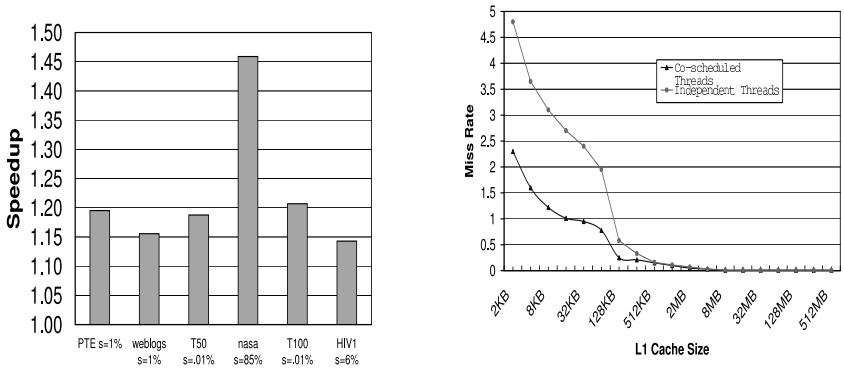

Figure 3: CMP code on a two way SMT (left), and working sets for co-scheduled threads vs independent threads (right).

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] 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.

[3] 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.

[4] A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y.-K. Chen, and P. Dubey. Cache-conscious frequent pattern mining on a modern processor. In *Proceedings of the 2005 International Conference on Very Large Databases (VLDB05)*, 2005.

[5] 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.

[6] R. Jin and G. Agrawal. An efficient association mining implementation on cluster of smps. In *Workshop on Parallel and Distributed Data Mining (PDDM)*, 2001.

[7] E.-Y. Kim, M. Busch, K. Abel, L. Fritts, P. Bustamante, J. Stanton, D. Lu, S. Wu, J. Glowczwskie, T. Rourke, D. Bogdan, M. Piatak, J. D. Lifson, R. Desrosiers, S. Wolinsky, and C. Miller. Retroviral recombination in vivo: Viral replication patterns and genetic structure of simian immunodeficiency virus (siv) populations in rhesus macaques after simultaneous or sequential intravaginal inoculation with sivmac239vpx/vpr and sivmac239nef. In *Journal of Virology*, volume 79, pages 4886 – 4895, 2005.

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

[9] 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.

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

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

[12] T. Meinl, I. Fischer, and M. Philippsen. Parallel mining for frequent fragments on a shared-memory multiprocessor. In *LWA 2005 - Beitrge zur GI-Workshopwoche Lernen*, pages 196–201, 2005.

[13] 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.

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

[15] 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.

[16] 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.

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

[18] Y. G. Sucahyo, R. P. Gopalan, and A. Rudra. Efficiently mining frequent patterns from dense datasets using a cluster of computers. In *Lecture Notes in Computer Science*, volume 2903, pages 233 – 244, 2003.

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

[20] 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.

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

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

[23] 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.

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