Parallel Mining for Frequent Fragments on a Shared-Memory Multiprocessor – Results and Java-Obstacles –

Thorsten Meinl, Ingrid Fischer, and Michael Philippsen
University of Erlangen-Nuremberg
Computer Science Department 2
Martensstr. 3, 91058 Erlangen, Germany
{meinl,idfische,philippsen}@cs.fau.de

Abstract

Although in the last years about a dozen sophisticated algorithms for mining frequent subgraphs have been proposed, it still takes too long to search big databases with 100,000 graphs and more. Even the currently fastest algorithms like gSpan, FFSM, Gaston, or MoFa need hours to complete their tasks.

This paper presents a thread-based parallel version of MoFa, [5] that achieves a speedup of about 7 on a shared-memory SMP system equipped with 12 processors. We discuss the design space of the parallelization, the results, the obstacles, that are caused by the irregular search space and by the current state of Java technology, and reason about ways to achieve even better speedups in future.

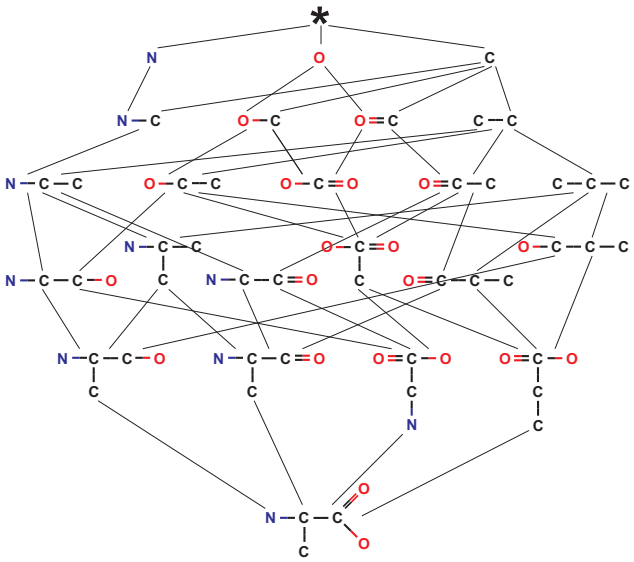

Figure 1: The complete fragment lattice of the molecule shown at the bottom.

1 Introduction

Mining of frequent subgraphs in graph databases is an important challenge, especially in its most important application area "chemoinformatics" where frequent molecular fragments help finding new drugs. Subgraph mining is more challenging than traditional data mining, since instead of bit vectors (i.e., frequent itemsets) arbitrary graph structures must be generated and matched. Since graph isomorphism testing is NP-complete [18], fragment miners are exponential in runtime and/or memory consumption. For a general overview see [12].

The naive fragment miner starts from the empty graph and recursively generates all possible refinements/fragment extensions by adding edges and nodes to already generated fragments. For each new possible fragment, it then performs a subgraph isomorphism test conceptually on each of the graphs in the graph database to determine, if that fragment appears frequently (i.e., if it has enough *support*). Since a new refinement can only appear in those graphs, that already hold the original fragment, the miner keeps appearance lists to restrict isomorphism testing to the graphs in these lists.

All possible graph fragments of a graph database form a lattice, see Fig. 1 for an example database with just one graph. The empty graph $*$ is given at the top, the final graph at the bottom of the picture. During the search this lattice will be pruned at infrequent fragments since their refinements will appear even more rarely. In the last few years sophisticated algorithms to solve this problem were presented [5; 28; 15; 22; 17], but still, the process of finding frequent fragments is (and is likely to remain) too time-consuming.

Although it may seem to be an obvious approach to split the problem into p parts, to solve the subproblems on p parallel processors, and to then hope for a speedup of p, only little work on parallel or distributed algorithms has been done in the area of frequent subgraph mining so far. The reason is that the problem is hard to parallelize. One issue is the highly irregular search space that requires sophisticated load balancing techniques. Especially for molecular databases, certain frequent fragments are bigger and have a higher support than others. Any search space partitioning (e.g., database partitioning) among the processors that does not take this effect into account must result in some processors that finish their work long before others whose part of the search space is more complex. Another issue is that the proper granularity of parallelization is far from obvious: On the one hand, almost each loop in the algorithm could be parallelized in a fine-granular way (which is common for e.g. parallel scientific Fortran programs); on the other hand, several coarse-grain "worker" threads could explore the search space. Finally, it is not straightforward to design optimal data structures since the performance implications are much more severe than for sequential algorithms. Either parallel activities that access shared data must be properly synchronized properly or replicated copies of the data must be kept in a consistent state. In both cases, runtime

costs must be considered. The potential runtime advantage of the latter approach is paid for by increased memory consumption.

Starting from the well-known sequential MoFa algorithm [5] (that is summarized in section 3) this paper derives and discusses a parallel algorithm for a shared-memory multiprocessor system in section 4. The state of currently available Java implementations on multiprocessors cause certain obstacle in the implementation; these are discussed in section 5.1. The performance measurements given in section 5.2 provides insights in ways to further improve the parallel algorithm.

2 Related Work

While general parallel depth first search strategies are well-known and quite easy to implement for many standard problems, for an introduction see for example [16], data or fragment mining requires special care.

The area of parallel or distributed data mining has been a hot topic for years. Related algorithms for parallel subgraph mining stem from the area of association rule mining. Just in the way subgraph miners search for frequent graph fragments, association rule miner search for sets of items that occur frequently in the database (e.g. market basket analysis). For the two well-known association rule miners Apriori [2] and Eclat [31] parallelized version have been developed, e.g. [3; 30; 29]. In contrast to our approach, these authors target cluster-like supercomputers that are interconnected by fast networks. Hence, these papers mainly deal with the different ways to distribute the work so that communication cost can be reduced. One extreme approach is to partition the database among the threads but to let all threads work on a single shared lattice. That approach is adopted in [8] for parallelizing the oldest graph mining algorithm *Subdue*. In the other extreme, the database is replicated, i.e., each thread has its own copy thereof while working on a private search lattice. More sophisticated methods to overcome the deficiencies of the early algorithms were presented in [13; 23]. In [9] the authors discuss the problems of data skewness and load balancing of association rule mining on a shared-nothing cluster. However, since the cost of network communication is not predominantly important on shared-memory machines, this issue need not be discussed thoroughly here. The only parallel/distributed implementation of MoFa is [11]. There the authors implemented a distributed version of the MoFa algorithm. The intended platform is a cluster of independent computers.

It seems to be more significant for this paper to focus on the two main difference between data mining and graph fragment mining, i.e., first, that the isomorphism test is much more expensive than the bit vector operations, and second, that fragment mining requires a lot more memory.

3 Finding frequent fragments with MoFa

Like many other subgraph miners, the MoFa-algorithm for finding frequent fragments in molecular databases [5] searches the lattice of all fragments in a depth-first way. New candidate fragments are created by extending existing frequent structures by an edge and a node (or only an edge if cycles are closed). In order to speed up candidate generation and frequency computation (i.e. counting the number of molecules a fragment occurs in) so-called *embedding lists* are used. An embedding is a map from the nodes and edges of a fragment to the corresponding nodes and edges in the molecules. An embedding list holds all possible embeddings into all molecules of the database. New fragments can easily be generated by looking at the surroundings of an embedded fragment in the molecule. In order to find out the number of supported molecules the embeddings list has to be scanned for all distinct molecules. Since this can be done in linear time, the use of embeddings greatly improves the overall performance. However, the obvious drawback is the huge memory requirement for storing all the embeddings. Especially for small fragments or fragments with symmetries, the number of embeddings can easily reach millions, even on small databases of a few thousand compounds.

As can be seen in Fig. 1, many structures can be reached following different paths through the lattice. This is undesirable, because theses duplicates have to be filtered out by expensive graph isomorphism tests. In order to reduce the number of these checks, MoFa applies the following set of pruning rules:

- **Frequency based pruning.** If a fragment is found during the search that does not have enough support in the database this branch of the search tree can be cut, because all other bigger fragments must be infrequent too. This is also known as the *frequency antimonotone constraint* from association rule mining [2; 31].
- **Structural pruning.** MoFa numbers the nodes in the structures according to the time when they are added to it. After extending a subgraph from node n, there are three different types of nodes in the subgraph: Nodes being added to the fragment *before* n may not be extended any more. Nodes being added to the fragment *after* n are freely extendable. The node n itself has only restricted extensibility, i.e., not all possible edge-node pairs may be added to it in the next step. Given a total order on node and edge labels (atoms are sorted by their numbers in the periodic system, edges are sorted by bond types), only those extensions are permitted, that are greater (with respect to that order) than the last extension.
- Finally, **equivalent sibling pruning** [6] removes children of a fragment that represent the same fragment as one of its siblings but that were extended at an atom with a higher index.

Of course, the pruning rules cannot avoid all duplicates. The remaining ones still need to be filtered out by checking for graph isomorphism in the already discovered structures.

When starting to address the parallel fragment mining problem, we had two main reasons to start from MoFa instead of starting from one of the other fast sequential fragment miners. First the basic algorithm of MoFa is much simpler than others. Second, several extensions to MoFa's base algorithm have been published, e.g., special handling of rings [14] to speed up the search even further, or the search for fragments with carbon chains of variable lengths [19]. The question, when to do the filtering will open up an interesting tradeoff in the parallel algorithm discussed next.

4 Parallelizing the search

Before parallelizing a sequential algorithm, one has the fundamental choice to either target a loosely coupled grid or cluster of several computers (*distributed* or *grid computing*) or to address a shared-memory multiprocessor ma-

chine (*SMP*, symmetric multiprocessing). While in general, SMP machines are much more expensive than clusters of workstations, the obvious advantage is their global shared memory that makes them not only easier to program but also often results in faster memory access, both in terms of latency and throughput. Therefore, in this paper we target the latter architecture. We present an implementation in Java mainly because of Java's built-in support for threads and synchronization that neatly fit this architecture.

Another fundamental decision is the granularity of parallelization. Because of almost negligible communication costs on SMP machines, one could try to parallelize as many parts of the algorithm as possible, e.g. like in parallel Fortran programs almost every loop with independent operations could be parallelized. However, because Java does not offer parallel loops but "only" threads that have to be implemented explicitly, this would render the code utterly complex and unreadable. A coarser granularity would for example parallelize the search for extensions and/or the filtering of duplicates (the graph isomorphism tests). This approach would however require many synchronization points because all threads are working on one (or several) global data structures. We favored a third alternative discussed next, namely to use a set of more or less independent workers.

4.1 General setup

For our implementation, we used a worker approach in which as many threads are started as there are CPUs in the machine. Each thread is a fully functional copy of MoFa with its own embedding lists, frequent fragment set, and stack of unprocessed nodes. This keeps the number of synchronization barriers at a minimum level. The only global data structures are the graphs of the database (which fortunately are read-only), a list of idle workers, and a global fragment set with which the workers merge their local sets. The general setup is depicted in Fig. 2. The search starts

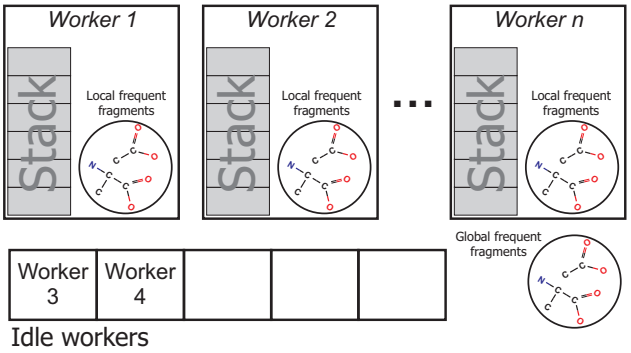

Figure 2: Key components of the parallel search

with the first worker searching for all frequent nodes and embedding them into all molecules of the database. Although the other workers are idle during that time, there is no need for the added complexity caused by a parallelization of this initial step, since it normally only takes a negligibly tiny fraction of the overall runtime.

Each one-node fragment forms a node in the search tree that consists of the fragment itself and its embedding list. They are all pushed onto the thread's local stack. Now, before the (next) call to the recursive search function, the running worker inspects the list of idle workers. If there is a waiting worker it is removed from the list. All list operations have to be synchronized among all threads since otherwise two running workers might remove the same idle thread.[1]

The interaction between the running worker and the idle worker can be achieved in two general ways. The easier-to-implement approach is to have the running worker *donate* part of its work to the idle worker by first splitting its stack of unprocessed search tree nodes into two halves and by then copying one part into the idle worker's stack. Alternatively, the idle worker can be woken up and actively *steal* work from the running worker. Work donation is much simpler to implement, since there is only one active thread. Therefore, no complex locking is required. In section 5.2 we will discuss the performance implications of our choice to use the donation approach.

This process is now repeated in every worker. Once a worker's stack becomes empty, it puts itself into the list of idle workers and waits for another worker to donate parts of its stack.

4.2 Load balancing

The straightforward way to implement work donation is to cut the stack in the middle. However, this creates a severe work imbalance among the two workers: The nodes on the bottom of the stack contain small fragments with long embedding lists and thus the number of children will be huge. On the other side, the nodes on the top of the stack consist of bigger fragments with shorter embedding lists and will therefore allow fewer extensions.

A perfect stack splitting would result in identical runtimes for both workers. Unfortunately, since there is no known cost function to calculate the precise runtimes required to process an unknown subgraph for each node on the stack, some heuristics need to be applied.

While it will be future work to conceive and quantitatively compare various stack splitting heuristics with respect to their overall runtime effects, a not-so-bad heuristics is *alternation-splitting*, i.e., to take every other element from the stack and transfer it to the idle worker's stack (see Fig. 3). Please note that because of this stack-splitting strategy the stack and recursion provided by the programming language cannot be used because entries in the stack except the top-most cannot be accessed. Thus an explicit stack and recursive search had to be implemented.

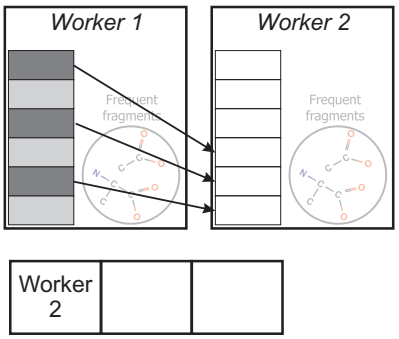

Figure 3: Alternation-splitting of the stack for better load balancing

4.3 Duplicate detection

As mentioned at the end of section 3, the locally applicable pruning rules cannot totally prevent the generation of

[1] It is convenient that the Java library provides a suitable list class with synchronized accessor methods.

duplicates.

If the workers would use a single shared fragment set, after generation of a fragment, it could immediately be checked whether that fragment has been generated before, maybe by another worker. At least conceptually, it would be necessary to lock the lattice whenever a worker modifies it so that no other worker can interfere. This would effectively turn the parallel workers into a sequentially processed sequence of set updates. Hence, all parallelism would be lost. There are of course better ways to do it: Instead of locking the whole set, locks of finer granularity can be used that would allow concurrent update operations on distinct areas of the set. The fact that the first step of duplicate detection is implemented by means of a hash table, leads to an approach that locks individual bins.

Since locking of the whole set is too inefficient and since we considered a fine-grained locking mechanisms to be too error-prone, in the current implementation each worker keeps its own local result set of frequent fragments. Of course, by keeping these sets separate, they might contain undetected duplicates. However, the increase of memory consumption by a factor of p turns out to be a significant problem, when p workers store fragments with their embedding lists redundantly. On the other hand, merging local result sets saves memory but causes a slowdown because of synchronization costs and – depending on the granularity of locks – by various degrees of sequentialization.

Our experiments show, that if only a small number (up to about 1,000) frequent fragments appeared in the data base, it is sufficient to let all the workers process until completion and then merge the local result sets in a final step. After all workers have finished their search, the master collects the structures in the local sets and merges them into the global set. Merging requires duplicate detection by costly isomorphism tests. (Instead of merging the local result sets one after the other sequentially, for large numbers of workers it might be beneficial to perform the merging in a a binary-tree based merging reduction. This has not been implemented yet, because the available SMP machine does not have enough CPUs to render this a profitable endeavor.)

In contrast, when searching with very low support values, several tens of thousand fragments are found. Among those where so many duplicates consuming so much memory that a final merge phase at the end was insufficient. In fact, since almost every worker had almost all fragments in its local result set, with p workers up to p times the heap space has been consumed that would have been necessary with a shared global set.

To circumvent the complexity of a parallel merging with fine-granularity locks, we have solved this problem in the following way: A maximum size for the local fragment sets is defined. If any of the workers reaches this threshold it tries to merge its local set with the global set. By this, most of the duplicates are already filtered out during the search. However, there is another issue with this approach. The merging of the sets takes quite some time, as many expensive graph isomorphism tests have to be made. Thus, if one thread is merging and another thread meanwhile reaches the limit, it is blocked because the access to the global set must be serialized. This is of course undesirable and therefore we chooses a lazy merging by using some kind of try-lock. The second thread first checks if the lock for the global fragment set is free and only in this case it acquires it. Otherwise it continues with the search and does not try to get the lock within a random number of iterations. This is important because with a very high probability the lock will be still in use the next time the thread tries to merge its local fragment set. As a side effect of this strategy the merging is implicitly done in parallel. Only after all workers have finished the remaining fragments in the local sets have to be added to the global set.

5 Experimental evaluation

We evaluated the performance of our parallel implementation on a cc-NUMA SGI Altix 3700 [25] equipped with 28 Itanium-2 (1.3 GHz) processors with a total of 112 GB RAM. The IA64-version of IBM's Java Development Kit 1.4.2 had to be used since Sun's JDK 1.4.2 was instable and frequently produced core-dumps. At the time of writing, the JDK 1.5 was not yet available for IA-64. The use of a 64-bit JVM is crucial because otherwise only about 3.5 GB of memory can be used. For all performance measurements the available heap has been set to 24GB.

Unfortunately, we could not get exclusive access to the Altix. Hence it was impossible to experiment with more than 12 CPUs. Moreover, we had to restrict the number of parallel garbage collector threads to 4. Otherwise, the JVM would have created 28 GC threads that overloaded the CPUs available to us.

5.1 Obstacles of current Java technology

The most severe difficulty when implementing the parallel MoFa in Java was that the Java virtual machine available to us has serialized all allocation and deallocation operations on the global heap. During the search MoFa allocates many small and short-living objects: Each found extension is represented as an object, each embedding consists of several objects, and each non-primitive data structure is an object as well. All these objects are allocated on the one and only global heap. Similarly, the concurrent garbage collector threads work on the same heap. The problem with current JVMs is that all heap modification operations are synchronized internally by the virtual machine so that only one modification is allowed at any given time. This led to situations where most of the workers were blocked waiting to get a chance to allocate an object on the global heap.[2]

To deal with this Java problem on multiprocessor machines, in version 1.4 so-called *thread-local allocation buffers (TLAB)* [20; 27] or *thread-local heaps (TLH)* [10; 24] have been introduced. The idea is to assign a private area of the heap to each thread that it can then access without acquiring a global heap lock. Only when a TLAB is full the thread has to access the global heap again. While TLABs might be a good idea for some applications, they are not suitable for parallel MoFa. The problem is that theses buffers are rather small in the standard settings (only some KB) and are filled up very quickly by MoFa. Even a manual increase of the buffer size did not improve the performance by much.

To solve this problem, each of the workers in our implementation uses a private *object pool* for the most frequently used objects, which are extensions and embeddings. Instead of directly creating an object by means of the `new`-operator, a request to the object pool is made. If the pool contains an unused object a reference to it is returned. If the pool is empty, it creates a new object. When an objects is not needed any more it is put back into the pool for future reuse. Although Sun officially discourages the use of

[2] We have verified this by means of the status dump functionality available in the IBM JVM implementation.

object pools [26] since future Java compilers will apply escape analysis [7] to allocate objects on a thread's runtime stack instead of the global heap whenever possible, at the time of writing only with private object pools we could see speedups at all – without object pools adding more workers even slowed down the total runtime.

Other than that, we had to implement the try-lock functionality by hand, since the JDK 1.4.x does not provide any library support for it. Such library support is not only available in the JDK 1.5, but in addition, the implementation makes use of hardware support for efficient non-blocking synchronization. Although our manual implementation is less efficient, we did not notice the difference in the benchmarks.

5.2 Performance measurements

Let us first study the performance of the parallel MoFa when searching for fragments that occur in at least 700 of all molecules (2% support) of the publicly available NCI Cancer (H23) dataset that consists of about 35,000 molecules [21]. The results for up to 12 workers are shown in Fig. 4 (the runtime is shown on the left x-axis, the speedup on the right). The following observations can be

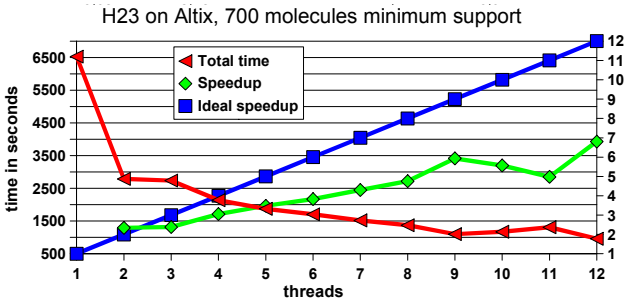

Figure 4: Runtime and speedup on the H23 dataset

made: First, on 12 CPUs the parallel version achieved a respectable speedup of about 7. Second, the speedup scales more or less linearly. The fact the the line is not smooth is mainly caused by our non-exclusive access to the Altix machine. When other users were working heavily our experiments slowed down a bit. However, there is room for improvement. Only the initial parsing of the molecules and the merging of thread local results are inherently sequential. These two sequential phases are very short as they take only about 5 of 958 seconds. The rest of the algorithm can, in principle, be done in parallel, except for synchronization requirements. When the cost of synchronization is ignored, Amdahl's law [4] predicts a potential speedup[3] of about 11 on 12 CPUs. A closer look at the benchmark results showed that too many workers were idle too often. The lack of work at the beginning has already been mentioned above. Since molecular datasets contain only a few frequent atom types (mainly carbon, nitrogen, oxygen, sulfur and sometimes chloron) the search starts with the embeddings of these four or five nodes into the whole dataset. Consequently, more than half of the workers are idle at the beginning. Only after extensions for the single-node fragments have been found, enough work will be available for all workers. Additionally during the search workers runs out of work. With the donation approach, such a worker has to wait until another worker finishes the extension of a fragment and can donate parts of its stack to the idle worker. Fig. 5 shows the sum of these idle times over all threads. It

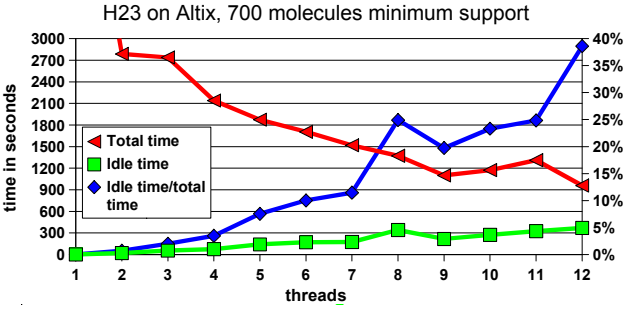

Figure 5: Idle times on the H23 dataset

can be clearly seen that the idle times rise steeply as more and more parallel workers are used. But keep in mind that this is the sum over all the idle times of all workers. With twelve threads almost forty percent of the total time could be saved if the idle times could have been avoided. This is a strong indication, that (a) better load balancing heuristics are needed that avoid workers becoming idle in the first place and (b) that the additional complexity of the work stealing approach might be worthwhile. We will further explore these issues in future.

With respect to Java's performance, it might be a reasonable idea to just wait a bit. From Java 1.1.5 to current Java 1.5 the SciMark benchmark [1] has seen an improvement by about a factor of 400 over the last 5 years. Since multicore platforms will make their way into the mainstream, it is very likely that JVM technology for shared-memory architectures will improve significantly in the near future.

6 Conclusions and Outlook

In this paper we presented a parallel implementation of the MoFa-algorithm for finding frequent fragments in graph databases. We used several independent workers that were represented by Java-threads. The results show that this approach scales linearly at least up to 12 parallel threads, where a speedup of 7 can be achieved. With current Java technology on shared-memory multiprocessors, significant workarounds are required to reach acceptable performance at all. However, the main performance problems are caused by too many workers being idle too often. Better load balancing is needed and it can be expected that work stealing will perform better than the simpler donation approach. In future, we will also study whether other algorithms for frequent subgraph mining that use less memory (like gSpan) might perform better.

References

[1] SciMark 2.0. http://math.nist.gov/scimark2/.

[2] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining Association Rules between Sets of Items in Large Databases. In Peter Buneman and Sushil Jajodia, editors, *Proc. 1993 ACM SIGMOD Int'l Conf. on Management of Data*, pages 207–216, Washington, D.C., USA, 1993. ACM Press.

[3] According to Amdahl, the maximum achievable speedup is bounded by the amount of sequential activities during the execution: $s_{max} = \frac{1}{F+(1-F)/N}$. F is the percentage of the algorithm that cannot be parallelized, $1 - F$ is the parallelizable fraction and N is the number of processors. Thus if N tends to infinity the maximum achievable speedup converges to $\frac{1}{F}$.

[3] Rakesh Agrawal and John C. Shafer. Parallel Mining of Association Rules. *IEEE Trans. on Knowledge And Data Engineering*, 8(6):962–969, December 1996.

[4] Gene Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, pages 483–485, 1967.

[5] Christian Borgelt and Michael R. Berthold. Mining Molecular Fragments: Finding Relevant Substructures of Molecules. In *Proc. IEEE Int'l Conf. on Data Mining ICDM*, pages 51–58, Maebashi City, Japan, November 2002.

[6] Christian Borgelt, Thorsten Meinl, and Michael R. Berthold. Advanced Pruning Strategies to Speed Up Mining Closed Molecular Fragments. In Wil Thissen, Peter Wieringa, Maja Pantic, and Marcel Ludema, editors, *Proc. of the 2004 IEEE Conf. on Systems, Man and Cybernetics, SMC 2004*, pages 4565 – 4570, Den Haag, The Netherlands, October 2004.

[7] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 1–19, New York, NY, USA, 1999. ACM Press.

[8] D. J. Cook, L. B. Holder, G. Galal, and R. Maglothin. Approaches to parallel graph-based knowledge discovery. *Journal of Parallel and Distributed Computing*, 61(3):427–446, 2001.

[9] Yongqiao Xiao David W. Cheung, Sau D. Lee. Effect of data skewness and workload balance in parallel data mining. *IEEE Transaction on Knowledge and Data Engineering*, 14(3):498–514, May/June 2002.

[10] Tamar Domani, Gal Goldstein, Elliot K. Kolodner, Ethan Lewis, Erez Petrank, and Dafna Sheinwald. Thread-local heaps for Java. *SIGPLAN Not.*, 38(2 supplement):76–87, February 2003.

[11] Giuseppe Di Fatta and Michael R. Berthold. Distributed Mining of Molecular Fragments. In Stat Matwin, editor, *IEEE Int'l Conf. on Data Mining, Workshop on Data Mining and the Grid*, pages 1–9, Edinburgh, UK, November 2004.

[12] Ingrid Fischer and Thorsten Meinl. Subgraph Mining. In J. Wang, editor, *Encyclopedia of Data Warehousing and Mining*, pages 1059–1063. Idea Group Reference, Hershey, PA, USA, July 2005.

[13] Eui-Hong Han, George Karypis, and Vipin Kumar. Scalable Parallel Data Mining for Association Rules. In *Proc. 1997 ACM SIGMOD Int'l Conf. on Management of Data*, pages 277–288, Tucson, AZ, USA, May 1997.

[14] Heiko Hofer, Christian Borgelt, and Michael R. Berthold. Large Scale Mining of Molecular Fragments with Wildcards. In *Advances in Intelligent Data Analysis*, number 2810 in Lecture Notes in Computer Science, pages 380–389. Springer, 2003.

[15] Jun Huan, Wei Wang, and Jan Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *Proc. of the 3rd IEEE Intl. Conf. on Data Mining ICDM*, pages 549–552, Melbourne, FL, USA, November 2003. IEEE Press.

[16] Vipin Kumar and V. Nageshwara Rao. Parallel Depth–First Search. *Int'l J. of Parallel Programming*, 16(6):501–519, December 1987.

[17] Michihiro Kuramochi and George Karypis. An efficient algorithm for discovering frequent subgraphs. *IEEE Trans. on Knowledge and Data Engineering*, 16(9):1038–1051, September 2004.

[18] Brendan McKay. Practical Graph Isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

[19] Thorsten Meinl, Christian Borgelt, and Michael R. Berthold. Mining Fragments with Fuzzy Chains in Molecular Databases. In Joost N. Kok and Takashi Washio, editors, *Proc. of the Workshop W7 on Mining Graphs, Trees and Sequences (MGTS '04)*, pages 49–60, Pisa, Italy, September 2004.

[20] Joseph D. Mocker. A collection of JVM options. http://blogs.sun.com/roller/resources/watt/jvm-options-list.html, May 2005.

[21] NCI. National Cancer Institute, DTP AIDS Antiviral Screen. http://dtp.nci.nih.gov/docs/aids/aids_data.html, March 1999.

[22] Siegfried Nijssen and Joost N. Kok. A Quickstart in Frequent Structure Mining can Make a Difference:. In Ronny Kohavi, Johannes Gehrke, William DuMouchel, and Joydeep Gosh, editors, *Proc. of the 10th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining (KDD2004)*, pages 647–652, New York, NY, USA, August 2004. ACM Press.

[23] Jong Soo Park, Ming-Syan Chen, and Philip S. Yu. Efficient parallel data mining for association rules. In *CIKM '95: Proc. of the fourth international conference on Information and knowledge management*, pages 31–36, New York, NY, USA, 1995. ACM Press.

[24] K. Kuiper R. Dimpsey, R. Arora. Java server performance: A case study of building efficient, scalable JVMs. *IBM SYSTEMS JOURNAL*, 39(1):151–175, 2000.

[25] Silicon Graphics, Inc. SGI Altix 3000. http://www.sgi.com/products/servers/altix/index.html, July 2005.

[26] Sun Microsystems, Inc. Frequently asked questions about the Java HotSpot VM. http://java.sun.com/docs/hotspot/PerformanceFAQ.html#15, July 2005.

[27] Sun Microsystems, Inc. Threading. http://java.sun.com/docs/hotspot/threads/threads.html, July 2005.

[28] Xifeng Yan and Jiawei Han. gSpan: Graph–Based Substructure Pattern Mining. In *Proc. IEEE Int'l Conf. on Data Mining ICDM*, pages 721–723, Maebashi City, Japan, November 2002.

[29] Mohammed J. Zaki. Parallel and Distributed Association Mining: A Survey. *IEEE Concurrency*, 7(4):14–25, December 1999.

[30] Mohammed J. Zaki, Srinivasan Parthasarathy, and Wei Li. A Localized Algorithm for Parallel Association Mining. In *ACM Symp. Parallel Algorithms and Architectures*, pages 321–330, Newport, RI, USA, June 1997.

[31] Mohammed J. Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, and Wei Li. New Algorithms for Fast Discovery of Association Rules. In David Heckerman, Heikki Mannila, Daryl Pregibon, Ramasamy Uthurusamy, and Menlo Park, editors, *Proc. of 3rd Int'l Conf. on Knowledge Discovery and Data Mining*, pages 283–296. AAAI Press, August 1997.