First publ. in: International Conference on Systems, Man and Cybernetics, 2006. SMC '06. Vol. 2, pp. 1269-1274

Mining Molecular Datasets on Symmetric Multiprocessor Systems

Thorsten Meinl
ALTANA Chair for Bioinformatics and Information Mining, University of Konstanz, Germany
meinl@inf.uni-konstanz.de

Marc Wörlein, Ingrid Fischer, Michael Philippsen
Computer Science Department 2
University of Erlangen-Nuremberg, Germany
{woerlein,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 thread-based parallel versions of MoFa [5] and gSpan [26] that achieve speedups up to 11 on a shared-memory SMP system using 12 processors. We discuss the design space of the parallelization, the results, and the obstacles, that are caused by the irregular search space and by the current state of Java technology.

I. Introduction

Mining for 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; instead of bit vectors (i.e., frequent itemsets) arbitrary graph structures must be generated and matched. Since subgraph isomorphism testing – which is a main part of subgraph miners – is NP-complete [15], fragment miners are exponential in runtime and/or memory consumption. For a general overview see [10].

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 (see [5], [11], [14], [19], [26]), 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 then to hope for a speedup of p, only little work on parallel or distributed algorithms has been done

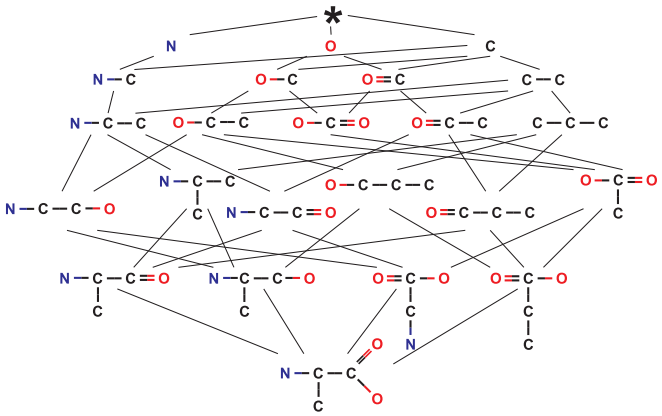

Fig. 1. The complete fragment lattice of the molecule shown at the bottom.

in the area of frequent subgraph mining so far. The reason is that the problem is hard to parallelize.

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 machine (*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.

In the literature no adaption of graph mining to SMP machines can be found. 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 [27] parallelized version have been developed, e.g. [12], [20]. For graph mining only parallelizations on PC clusters with distributed memory have been developed for Subdue [7] and MoFa [9].

In the following sections we present our parallelized implementations of the two subgraph miners MoFa and gSpan. In section II we shortly describe the sequential algorithms, their commons and their differences. The main part in section III deals with the design of the parallel versions and what advantages and disadvantages our choices may have. As

MoFa and gSpan differ substantially in their use of memory and CPU-intensive calculations, different approaches have to be used. Experimental results are discussed based on design choices for the parallelization and algorithmic properties of the underlying algorithms in section IV.

II. The MoFa and gSpan algorithms

Two of the most popular subgraph miners are MoFa [5] and gSpan [26]. They produce the same results but their algorithms differ substantially algorithmically.[1] Subgraph miners can be classified by many properties, the most important for the difference between gSpan and MoFa are:

- **Search strategy.** The way in which the subgraph lattice is traversed can be either depth- or breadth first. Depth-first search, as used by MoFa and gSpan, requires less memory because only few fragments have to be stored in memory (proportional to the length of the longest path). On the other hand breadth first search allows for more efficient search space pruning because the frequency information of all possible ancestors is available.
- **Candidate generation.** New fragments have to be generated in some way from the already found ones. Both algorithms extend fragments by adding an edge and a node (except for cycle-closing edges). This requires access to the database because all possible extensions have to be determined somehow.
- **Support computation.** For each generated candidate fragment, its support (i.e. the number of graphs it occurs in) has to be determined. This can either be done by explicit subgraph isomorphism tests or by using embeddings lists that store information about the exact occurrences of the ancestor fragment in the database. gSpan relies on explicit subgraph isomorphism tests but interleaves the counting with the search for possible extensions of the ancestor fragment. MoFa on the other hand uses embedding lists and avoids the explicit subgraph isomorphism tests. The support can simply be determined by counting the distinct graphs referred to by the embeddings in the list. This, however, comes at the cost of huge memory requirements because essentially *all* subgraph isomorphisms are stored in the embedding lists and their number can be quite large.
- **Duplicate elimination.** From the lattice structure of all subgraphs it becomes clear that there are many possible ways to reach a certain fragment. The algorithms try to prune as many redundant paths out of the lattice as possible. gSpan achieves this by using a canonical form for the representation of graphs. It extends fragments only at special paths that are easy to identify based on the canonical form. MoFa only allows extensions at the latest extended node and nodes that have been added after this node. Unfortunately, these powerful pruning techniques cannot avoid all duplicates; the remaining ones have to be filtered out. MoFa stores all fragments found in a list and for every new fragment a graph isomorphism test is done against the members of the list. For gSpan, subgraphs that have not been built in the order defined by the canonical form are duplicates. This has some advantages over the graph isomorphism tests because duplicates can be found by just "locally" looking at the fragment itself and there is no need for a "global" list of all discovered frequent subgraphs.

A detailed comparison of the sequential algorithms can be found in [25]. In this paper we are interested in how the different approaches affect the parallelization. This will be discussed in the next section.

III. Parallelizing the search

As mentioned in the introduction, the search for frequent fragments is hard to parallelize. One problem is the highly irregular search space that requires sophisticated load balancing techniques. Especially for molecular databases, some frequent fragments are bigger or 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 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.

We present implementations in Java mainly because of Java's built-in support for threads and synchronization that neatly fit this requirements and also supports the selected SMP architecture.

The choice of Java threads almost automatically determines the granularity of the parallelization. Unlike e.g. Fortran, where each loop can easily be parallelized, Java threads are more suitable for a coarser granularity where large parts of the search are done in several more or less independent "workers". Thus our general setup for both algorithms looks as shown in Fig. 2. Each thread is a fully functional copy of either MoFa or gSpan with its own 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 workers, and a global fragment set with which the workers merge their local sets. The search starts with the first worker looking for all frequent

[1] The research in this paper was done with the original version of MoFa as described in [5] and not the modified one presented in [4].

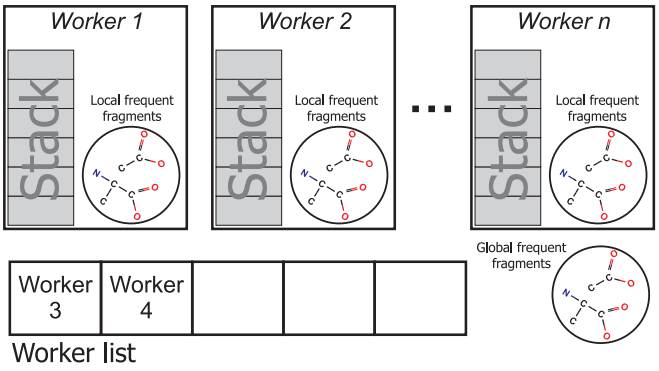

Fig. 2. Key components of the parallel search

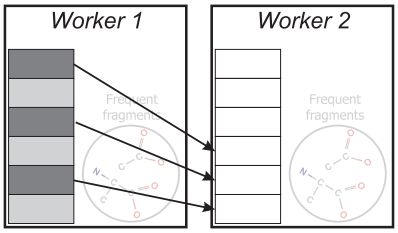

Fig. 3. Alternation-splitting of the stack for better load balancing

nodes. 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.

So far the setup is more or less identical for both MoFa and gSpan. They differ, however, in the way the work is then spread over the the available workers/threads.

A. Load balancing

For both algorithms stack based dynamic load balancing is used. Therefore each thread works on its local stack as long as there is work available. If a stack runs empty, it will be refilled by the values of another non-empty stack (held by another thread).

It is too naive to simply cut a stack into halves, i.e., to move the first $\frac{n}{2}$ elements to the empty stack and to keep the second $\frac{n}{2}$ at the current thread. This approach results in imbalances since the stack just refilled will run empty again soon. Search tree nodes near the bottom of a stack are more labor intensive than elements on top of the stack, because the small fragments found in the early stages of the search allow for much more extensions. A perfect load balance would be achieved if the exact amount of work for each node could be determined in advance. This is almost impossible for frequent fragment mining. Thus we have chosen another quite simple scheme introduced in [13] that performs much better than just cutting the stack into two halves: The “first” thread gets all odd elements of the stack, the “second” one all even elements, see Fig. 3.

For MoFa we have chosen a so-called *work donation* scheme to distribute and balance the work among the threads. With work donation the global list is populated with *idle* threads. Each time a running worker finishes an iteration of its main loop, i.e. it has found extensions of a fragment and puts them onto its local stack, it checks if there are any idle workers waiting in the global list. If so, it takes the first one and donates half of its local stack to the idle thread. The workers continue searching with the elements on their local stacks.

The advantage of this work donation scheme is, that it requires only a single synchronized data structure: the global list of idle workers. Only if a thread runs out of work and puts itself into the list or if a running thread checks for workers in the list a synchronization has to be done. On the other hand, work donation may not use the full power of the system. Especially if the running threads have a lot of work to do (e.g. extending fragments with lots of embeddings) it takes some time until one of the threads takes a look at the idle list and donates parts of his stack to one of the idle workers. CPU time can be wasted.

Therefore we tried another technique, referred to as *work stealing* for gSpan. Idle threads do not wait passively for new work. They actively “steal” the work of other running threads to refill their own empty stacks. To get access to all non-empty stacks, a global worker list of *active* workers is needed (instead of one for *idle* workers). If one thread runs empty, it first removes itself from that global list, and afterwards iteratively tries to split the non-empty stacks of the remaining (running) workers on the list to get half of their work. The same splitting scheme as above is used.

The benefit of this technique is, that the stack splits do not stress the working threads. This task is done by the “idle” workers, so no CPU time is wasted like in work donation. Also, as long as there is work available for each thread no checks for idle threads have to be performed. On the other hand, during the splitting of a stack, the “idle” and the active thread concurrently access the same stack so additional synchronization for the local stacks is required.

B. Duplicate detection

As mentioned in section II, the generation of duplicates cannot be avoided completely and thus duplicates have to be filtered out.

For gSpan duplicate detection is straight forward using the canonical form. If a graph is not represented by its canonical form, it is a duplicate. So each thread can use a local set for its found fragment. These sets are put together in the end with almost no computational effort.

In MoFa each new fragment has to be checked against all previously found fragments. In a single shared fragment set for all workers it could immediately be checked whether a newly generated fragment has been inserted before, maybe by another worker. At least conceptually, it is necessary to lock this global set 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. Instead of locking the whole set, locks of finer granularity can be used allowing for 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 MoFa implementation each worker keeps its own local result set of frequent fragments. Of course, by keeping these sets separate, they might contain undetected duplicates. This will turn into a severe problem, if the number of fragments gets large, because more and more duplicate fragments in the local sets will occur.

Our experiments showed, 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.[2]

In contrast, when searching with very low support values, several tens of thousand fragments are found. Among those where so many duplicates that consumed 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 choose 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.

IV. Experimental evaluation

We evaluated the performance of our parallel implementations on a cc-NUMA SGI Altix 3700 [22] equipped with 32 Itanium-2 (1.6 GHz) processors and 122 GB RAM. The only available JVM was the IA64-version of IBM's Java Development Kit 1.4.2 and the available heap size has been set to 60 GB for each run. Unfortunately, we could not get exclusive access to the Altix machine. 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 32 GC threads that overloaded the CPUs available to us.

A. Obstacles of current Java technology

The most severe difficulty when implementing the parallel miners 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 many small and short-living objects are created: Each found extension is represented as an object, and each non-primitive data structure is an object as well, even each stored or temporal embedding. 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.[3]

To deal with this Java problem on multiprocessor machines, in version 1.4 so-called *thread-local allocation buffers (TLAB)* [16], [24] or *thread-local heaps (TLH)* [8], [21] 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 subgraph miners. The problem is that theses buffers are rather small in the standard settings (only some KB) and are filled up very quickly by MoFa and gSpan. Even a manual increase of the buffer size did not improve the performance much.

To solve this problem, each of the workers in our implementations uses a private *object pool* for the most frequently used objects: 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 object is not needed any more it is put back into the pool for future reuse. Although Sun officially discourages the use of object pools [23] since future Java compilers will possibly apply escape analysis [6] 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.

[2]Because 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 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.

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

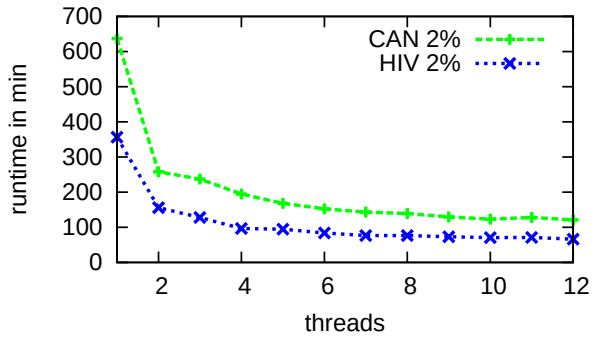

Fig. 4. Runtime for MoFa on the CAN and HIV dataset

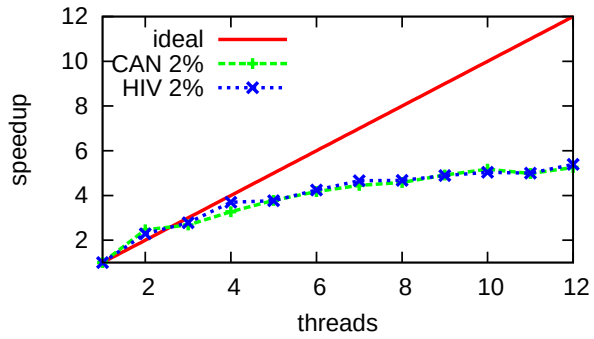

Fig. 5. Speedup for MoFa on the CAN and HIV dataset

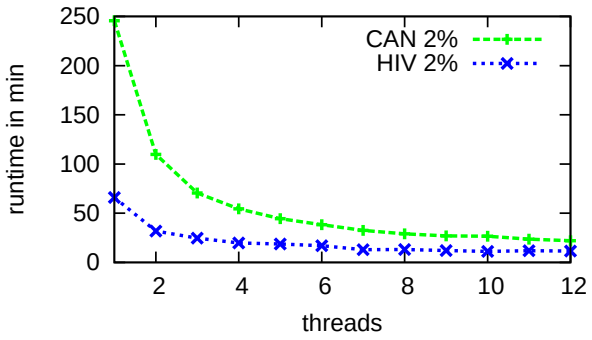

Fig. 6. Runtime for gSpan on the CAN and HIV dataset

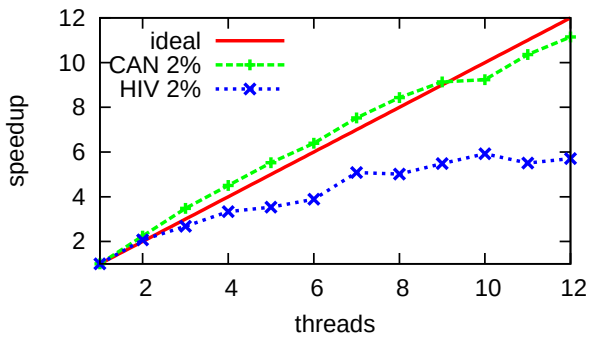

Fig. 7. Speedup for gSpan on the CAN and HIV dataset

B. *Performance measurements*

We searched for fragments that occur in at least 2% of all molecules of the publicly available NCI Cancer (CAN) and NCI HIV datasets that consist of about 35,000 [18] and 42,000 molecules [17], respectively. The runtimes and speedups for up to 12 workers using MoFa are shown in Fig. 4 and Fig. 5.

The following observations can be made: First, on 12 CPUs the parallel version achieved a speedup of about 5.5. Second, the speedup scales more or less linearly. The fact that the line is not smooth is mainly caused by our non-exclusive access to the Altix machine. When other users were working, our experiments slowed down. 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 $4-5$ 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 [3] predicts a potential speedup of about 12 on 12 CPUs.[4]

After the look at that MoFa results, the same tests on the Cancer and HIV dataset are made with the gSpan implementation. The runtime is given in Fig. 6 and the speedup in Fig. 7. For the HIV dataset a lower linear speedup compared to the Cancer dataset can be observed. Up to 10 threads the speedup for the HIV dataset increases linearly, for 11 and 12 CPUs it decreases again. This can be explained by the high number of embeddings in the HIV dataset and the fact that gSpan does not store embeddings but generates them again. Multiple temporal objects not stored in the object pools (like local arrays) have to be generated to re-detect the embeddings. The influence of the synchronized Java heap hurts databases with many potential embeddings more than others. As more threads are working more time is wasted with the synchronization and the overall time of the search slows down.

For 2 to 9 threads the speedup is more than ideal for the Cancer dataset. This is very likely a result of the nonexclusive access to the test machine. The experiments with one thread ran very long and other users could influence the results negatively even more. So the reference measurement is slightly worse it normally would be and thus the speedup of the runs with more threads rises of course.

gSpan tops the results of MoFa. This can be explained again through the less memory consumption of the gSpan algorithm. One the one hand, as no embeddings are stored, the corresponding objects can be reused more regularly than for MoFa. Because of the non-synchronized object pool less synchronized heap accesses are necessary which leads to a better speedup. On the other hand not only the heap is used more heavily but also the uncommon memory architecture of the used Altix machine may influence the MoFa algorithm

[4]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 $\frac{1}{F}$.

more severely than gSpan. This is, however, a problem that goes beyond the scope of our research and another SMP architecture may show better results.

Last but not least we have also checked how the different load balancing techniques (work donation or stealing) influence the scalability of the algorithms. In general work stealing is assumed to scale better than work donation, which was also explained in section III-A. Therefore we summed up the "dead-times", i.e. the time a worker waits in the idle list. However, for MoFa's work donation approach the dead times were in the range of a few seconds only (gSpan does not have any dead time whatsoever). In relation to the overall runtime this is at most one percent and thus their influence is neglibible. It seems that (at least for the used datasets and parameters) there is no significant advantage of using a work stealing approach.

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 five years. Since multi-core 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.

V. Conclusions

In this paper we presented parallel implementations of the two subgraph miners MoFa and gSpan. 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 up to 11 can be achieved. With current Java technology on shared-memory multiprocessors, significant workarounds are required to reach acceptable performance at all.

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

[4] Christian Borgelt. On canonical forms for frequent graph mining. In Siegfried Nijssen, Thorsten Meinl, and George Karypis, editors, *3rd International Workshop on Mining Graphs, Trees, and Sequences (MGTS'05 at PKDD'05)*, pages 1–12, Porto, Portugal, October 2005.

[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] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] Ruoming Jin, Ge Yang, and Gagan Agrawal. Shared memory parallelization of data mining algorithms: Techniques, programming interface, and performance. *IEEE Trans. on Knowledge and Data Engineering*, 16(19):71–89, October 2004.

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

[14] 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.

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

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

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

[18] NCI. National Cancer Institute, DTP Human Tumor Cell Line Screen. http://dtp.nci.nih.gov/docs/cancer/cancer_data.html, March 1999.

[19] 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.

[20] Srinivasan Parthasarathy, Mohammed Javeed Zaki, Mitsunori Ogihara, and Wei Li. Parallel data mining for association rules on shared-memory systems. *Knowl. Inf. Syst.*, 3(1):1–29, 2001.

[21] 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.

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

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

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

[25] Marc Wörlein, Thorsten Meinl, Ingrid Fischer, and Michael Philippsen. A quantitative comparison of the subgraph miners MoFa, gSpan, FFSM, and Gaston. In Alipio Jorge, Luis Torgo, Pavel Brazdil, Rui Camacho, and Joao Gama, editors, *Knowledge Discovery in Database: PKDD 2005*, Lecture Notes in Computer Science, pages 392–403, Berlin, 2005. Springer.

[26] 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.

[27] 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.