Possibilities to solve the clique problem by thread parallelism using task pools

H. Blaar, M. Karnstedt, T. Lange, R. Winter

Martin-Luther-University Halle-Wittenberg, Institut of Computer Science
06099 Halle (Saale), Germany

Abstract

We construct parallel algorithms with implementations to solve the clique problem in practice and research their computing time compared with sequential algorithms. The parallel algorithms are implemented in Java using threads. Best efficiency is achieved by solving the problem of task scheduling by using task pools.

Key words: clique problem, parallelism, thread, task pool

1 Introduction

1.1 Aim

In accordance with the hypothesis of parallel computing all problems solvable in polynomial space have a parallel algorithm with polynomial time complexity. Efficient parallel algorithms exist for NP-complete problems [13]. Theoretically an exponential number of processors delivers polynomial time to solve the clique problem. But in practice we do not have so many processors. In this article we investigate parallel algorithms in reality and develop suitable possibilities to solve the NP-complete clique problem by parallelism in practice.

1.2 Fundamental definitions and basic foundations

A clique of a graph G is a set V' of pairwise adjacent vertices in G. The clique number $\omega(G)$ of G is defined as the size of a largest clique contained in G.

Email address: `[blaar, winter]@informatik.uni-halle.de` (H. Blaar, M. Karnstedt, T. Lange, R. Winter).

The clique decision problem is given as follows:

Definition 1.2.1: $CLIQUE1(G, k)$ *(decision problem)*
Instance: Graph $G = [V, E]$ *with a set* V *of vertices and a set* $E \subseteq V \times V$ *of edges,* $|V| = n, k \in N, k \leq n$.
Question: Does G *have a* k*-clique, i.e., does exist a set* $V' \subseteq V$ *with* $|V'| = k$*, so that for every pair* $v_1, v_2 \in V'$ $(v_1, v_2) \in E$ *?*

Definition 1.2.2: $CLIQUE2(G)$ *(number problem)*
Instance: Graph $G = [V, E]$.
Problem: Compute the maximal k *(denoted* $\omega(G)$*) for* $V' \subseteq V$ *with* $|V'| = k$.

Definition 1.2.3: $CLIQUE3(G)$ *(optimization problem)*
Instance: Graph $G = [V, E]$.
Problem: Compute the maximal clique $V' \subseteq V$.

$CLIQUE1$ is a NP-complete problem. It is not known a polynomial time algorithm for solving this problem. The complexity does not rest on the size of numbers. Extremely large numbers would be unlikely to occur. We can use 1 to n for n nodes. A limitation of the size of numbers is not profitable. $CLIQUE$ is strong NP-complete. There do not exist pseudopolynomial algorithms unless $P = NP$ [14]. The PCP-theory proofs the clique problem as not solvable by approximation with reasonable results [9]. The strong NP-completeness and the hopeless outlook for a good result by means of approximation show the hardness of finding cliques in graphs by sequential algorithms.

Unlike in theoretical we can solve the clique problem in polynomial time on parallel computers. If a problem is sequential solvable in space $s(n)$ with input length n so it is parallel solvable in time $s(n)$. Considering a Turing machine as model of a sequential computer and a PRAM as model of a parallel computer, (1)–O–TM-SPACE(s) is the family of problems decidable on a 1-tape deterministic Turing machine with space complexity $s(n)$, $PRAM$–TIME(t) is the family of problems decidable on a PRAM with time complexity $t(n)$ [13].

Theorem 1.2.1 (1)–O–TM-SPACE(s) $\subseteq$ $PRAM$–TIME($\Theta(s)$) *for* $s \geq id_N$ *with identity function* $id_N(n) = n$. *[13]*

With NP as the family of problems acceptable of a nondeterministic Turing machine with polynomial time complexity we get

Conclusion 1.2.2: $NP \subseteq PRAM$–TIME($Pol(id_N)$).

Conclusion 1.2.3: $CLIQUE1 \in PRAM$–TIME($Pol(id_N)$).

Each NP-complete problem - also the clique problem - is efficiently paralleliz-

able if we have a sufficient number of processors.

1.3 A rough concept

In polynomial time we decide by means of a sequential algorithm whether the graph G or the complement $\overline{G}$ is for instance a tree, forest, perfect, chordal or bipartite graph (see [2], [12]), and get a polynomial sequential algorithm for CLIQUE1, CLIQUE2 and CLIQUE3 in such a case. Otherwise we do the following:

> If possible, we allocate the $\binom{n}{k}$ subsets of vertices to $\binom{n}{k}$ processors, which proof in parallel the pairwise connection of their vertices.
> In other cases we use a parallel greedy algorithm or parallelize the algorithm of Bron and Kerbosch. [4]

2 Graphs to test parallel algorithms

2.1 Graphs with easy computable cliques

For trees, perfect, chordal or bipartite graphs G we solve $CLIQUE1(G,k)$, $CLIQUE2(G)$, $CLIQUE3(G)$, $CLIQUE1(\overline{G},k)$, $CLIQUE2(\overline{G})$, and $CLIQUE3(\overline{G})$ in polynomial time by a sequential algorithm (see [6]). In such a case we do not need parallelism.

2.2 Graphs with cliques hard to compute

We generate and observe random graphs and graphs with a great number of maximal cliques to see difficulties of the sequential greedy and sequential Bron and Kerbosch algorithm. For such graphs a parallel algorithm is practical.

Definition 2.2.1 *A felt graph $G_n = (V_n, E_n)$ of complexity n contains n components $S_1, S_2, ..., S_n$ with*
(1) every component S_i is a circle with 5 nodes, $1 \leq i \leq n$,
(2) $\forall i \forall j \forall u \forall v (1 \leq i \neq j \leq n \wedge u \in V_i \wedge v \in V_j \rightarrow (u,v) \in E_n)$.

Definition 2.2.2 *A Moon-Moser graph $G_n = (V_n, E_n)$ of complexity n contains n components $S_1, S_2, ..., S_n$ with*
(1) every component S_i has 3 isolated nodes, $1 \leq i \leq n$
(2) $E_n = \{(u,v):\ u \in S_i \wedge v \in S_j \wedge i \neq j\}$.

Definition 2.2.3 *A Tarjan graph $G_{m,n} = (V,E)$ contains $2n$ crowds K_1^0, K_1^1, K_2^0, $K_2^1, ..., K_n^0, K_n^1$, each with m nodes. Two nodes $u \in V_i^x$ and $v \in V_j^y$ are not adjacent if $i = j$ and $x \neq y$.*

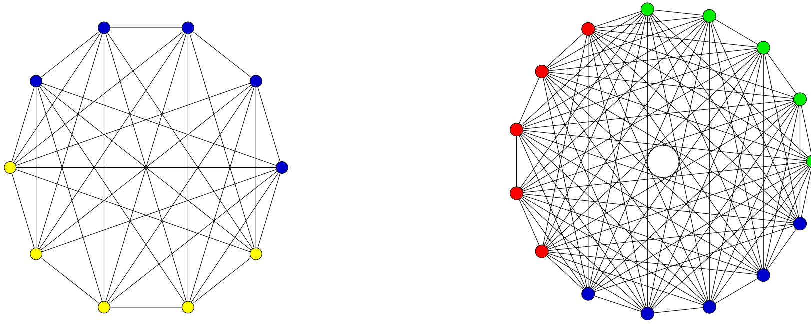

Fig. 1. Felts with complexity 2 and 3.

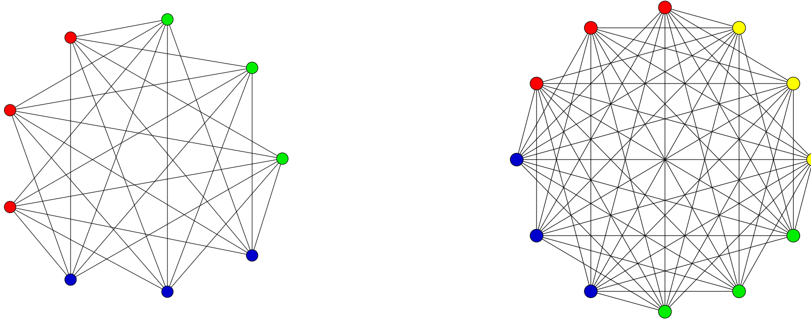

Fig. 2. Moon-Moser graphs with complexity 3 and 4.

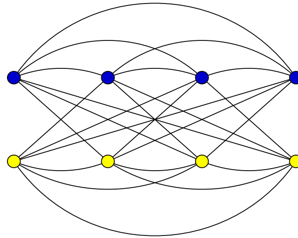

Fig. 3. Tarjan graph $G_{m,4}$. Every node in $G_{m,4}$ is a crowd with m vertices.

Conclusion: *For all* $n > 0$

(1) a felt G_n *contains* 5^n *cliques,*
(2) a Moon-Moser graph contains 3^n *cliques,*
(3) a Tarjan graph contains 2^n *cliques.*

3 Thread parallelism and task pools

To test our parallel algorithms we use a shared memory system. On such systems we have a common memory and a global address space. In our algorithms the amount of data exchange between parallel tasks is highly compared with the computing time. In addition a balanced task scheduling is difficult in graph

algorithms of this type. So on distributed memory systems the communication cost would be high, and a balanced task distribution very complex and cost-intensive, too. Using threads to implement parallel algorithms should be a suitable method for parallel solving the clique problem. Threads are subprocesses, started by a process, which use the context of this process. That means, the threads of one program have a common address space. The programmer has to synchronize the parallel work of the threads. The scheduling of tasks so that all processors can work without long waiting time is a problem with impact on efficiency of a parallel implementation. Using task pools several processors can work in balance without waiting time. In this section we give a short view of Java threads, task pools, and the runtime of multithreaded programs.

3.1 Java threads

Java has a direct support for multithreading integrated in the language, see, e.g., [10]. This is in contrast to most other programming languages where the operating system and a specific thread library like Pthreads [3] or C-Threads are responsible for the thread management. Threads can be generated by specifying a new class which inherits from the class `Thread` of the `java.lang` package and by overriding the `run()` method in the new class with the code that should be executed by the new thread. Then a new thread is created by generating an object of the new class and calling its `start()` method. The Java thread API offers an alternative way to generate threads by using the interface `Runnable` which contains only the abstract method `run()`. The `Thread` class actually implements the `Runnable` interface and, thus, a class inheriting from `Thread` also implements the `Runnable` interface. The method `setPriority(int prio)` of the `Thread` class can be used to assign a priority level between 1 and 10 to a thread where 10 is the highest priority level. The priority of a thread is used for the scheduling of the user level threads by the thread library, see below.

Access to the same data structures by several threads has to be protected by a synchronization mechanism. Java supports synchronization by implicitly assigning a lock to each object. The most easy way to keep a method thread-safe is to declare it as `synchronized`. A thread must obtain the lock of an object before it can execute any of its `synchronized` methods. No other thread can execute any other `synchronized` method of this object at the same time. An alternative synchronization mechanism is provided by the `wait()` and `notify()` methods of the `Object` class which also supports a list of waiting threads. Since every object in the Java system inherits directly or indirectly from the `Object` class, it is also an `Object` and hence supports this mechanism. These basic synchronization primitives can be used to realize more complex synchronization mechanisms like barriers, condition variables,

semaphores, event synchronization, and monitors, see, e.g., [10].

Thread creation and management are integrated into the Java language and runtime system and thus are independent of a specific platform. The mapping of the threads that are generated by a user program (user level threads) to the kernel threads of the operating system depends on the implementation of the Java Virtual Machine (JVM). The most flexible multithreading model is the many-to-many model which is, e.g., realized in SOLARIS Native Threads provided by the SOLARIS 2.x operating system. This model allows many threads in each user process. The threads of every process are mapped by the thread library to the kernel threads of the operating system which are then mapped to the different processors of the target machine by the scheduler of the operating system. The intention of this two-layer thread model is to decouple the scheduling and management of user threads from the kernel. SOLARIS uses lightweight processes (LWPs) to establish the connection between user threads and kernel threads. User threads have their own priority scheme and are scheduled with a separate scheduling thread that is automatically created by the thread library. For a detailed description see [8].

The difficulty of experiments with Java threads lies in the fact that the scheduling is not under the control of the programmer. When starting a thread-parallel program, it is not clear which thread will be assigned to which LWP and which LWP will be executed on which processor. The number of LWPs can be set to a specific value, but the operating system can adapt this number during the execution of the program. The number of processors used for the execution of a program is completely under the control of the operating system and can vary during the execution of a thread-parallel Java program.

3.2 Task pools

Independent or concurrent parts of an algorithm or program can be taken as *tasks*, and so we have *task parallelism*. To exploit task parallelism the tasks can be represented in a *task graph*. The nodes correspond with the tasks, and the edges represent the dependencies between the tasks. To assign the tasks to threads (or processors) scheduling methods can be used. With *task pools* a dynamical scheduling is executable. A task pool holds several executable tasks. If one thread has finished its task a new task from the task pool is assigned to it. With a finished task all of its predecessors with completed successors in the task graph will become executable and can be included into the task pool. During executing an algorithm, such as solving clique problem, new tasks can be generated dynamically. By varying the number of tasks in a task pool the overhaed of managing this method can be influenced. There should be enough executable tasks in the pool to avoid waiting periods of threads. On the other hand too much tasks will produce a high overhead. To find a good

balance is difficult because of the unknown dependencies between tasks in many problems. Task pools can be realized, for example, by *task queues*. In a *central* task queue all tasks are managed centrally, for example by one thread (or processor). Here the access to the queue must be synchronized in a way, that only one thread can access the queue at the same time. In a *decentralized* task-queue implementation every thread (or processor) holds a separate task queue and creates and finishes its own tasks. For more details see, e.g., [11], [7]. The structure of our implementation of the used task pool is shown in Figure 4.

```
public class ThreadPool  {              // task pool implementation
   Vector tasks;
      // ...
   ThreadPoolThread poolThreads[];
      // ...
   class ThreadPoolThread extends Thread  {   // working threads
      // ...
      public void run()    { //...
      }
   public void addRequestAndWait(Runnable target) // method to put
                    throws InterruptedException { // tasks into
      // ...                             the task pool (synchronized)
   }
   public void waitForAll(boolean terminate)       // synchronized
                    throws InterruptedException  { // termination
      // ...
   }
}
```

Fig. 4. Java code fragment: structure of the task pool implementation.

3.3 Runtime of multi-threaded programs

For measuring the runtime of a parallel application, the *real time* (wall clock time) of an application can be used, but it can be influenced by other applications running on the system. The user and system CPU time of a parallel application is the *accumulated* user and system CPU time on all processors, i.e., to compute the actual parallel CPU time, these values have to be divided by the number of processors used. Because of the influence of other processes running on the system the number of processors used for a parallel application cannot easily be determined and can also vary during the execution of a program. Thus, we try to use an unloaded system for real-time measurements to avoid the influence of other processes.

To get an impression of the overhead of creating, managing, and deleting a thread, we have performed runtime tests (see [1]). The results show, that the scheduling of the user threads does not cause a significant overhead whereas

the scheduling of the LWPs causes runtime costs that increase considerably with the number of LWPs, since the operating system is involved. This suggests to adjust the number of LWPs to the number of processors available.

4 Parallelization and implementation of clique problem algorithms

4.1 A parallel greedy algorithm

The following greedy algorithm decides $CLIQUE1(G, K) =$'true' exactly. If the algorithm is resulting in $CLIQUE1(G, k) =$'false' we can not guarantee the correctness of this result and so have to use another algorithm.

(1) $L_{Cl} = \emptyset$ (L_{Cl} is the set of produced clique nodes)
    $L = V$ (L is the set of nodes to choose from, beginning by V)
(2) $L_0 = L, v_0 = ZERO$
(3) Choose from $L \setminus L_{Cl}$ a node v with maximal degree.
    If $v_0 = ZERO$ store v, i.e., $v_0 = v$.
    $L_{Cl} = L_{Cl} \cup \{v\}$.
    If does not exist a node v with degree $\geq k - 1$, then does not exist a k-clique in G. STOP
    If $|L_{Cl}| = k$ we have found a k-clique. STOP
    Else to step 4.
(4) Compute L as the set of all neighbors of v. If $L_{Cl} \setminus \{v\} \not\subseteq L$, go to the next step.
    Else to step 3.
(5) $L = L_0 \setminus \{v_0\}, L_{Cl} = \emptyset$ and go to step 2.

We can obtain a parallel greedy algorithm for $CLIQUE1(G, k)$ simply by extracting as much parallelism as possible from the sequential one in step 3. If we start a thread for every parallel step we get "out of memory" and a large overhead. To avoid this effect it is better to use a task pool with constant size. But also in this case the time to manage the threads exceeds the time to real computation.

4.2 A simple parallel algorithm - verification of node subsets

We present a subset $V' \subseteq V$ as a vector $\in \{0, 1\}^n$ with number 1 at k positions. By shifting positions we get all subsets V' with k nodes. Looking at the adjacent matrix of G the produced subsets are proved to be a clique or not. Speed up we get by clever sharing of coasts. In the ideal case each processor has one subset V' to prove on completeness. We consider two possibilities of parallelism:

(1) One thread enumerates and shares out all subsets in uniform packages. Every processor gets a package and inspects the pairwise connection of the given nodes.
(2) A central thread activates the other threads. These threads compute the own subset and check it on clique quality.

To get a minimal overhead and no space problems we choose the number of employed threads equal to the number of existing processors. On the other hand for better working to capacity of the processors we can place the jobs into a task pool and use more threads than processors. Table 1 holds several computation results with 8 processors on a SunFire 3800.

Graph [# nodes]	sequential	parallel	
		without task pool	with task pool
random [60]	13.16 sec	34.63 sec	22.25 sec
felt [30]	1 min 27 sec	1 min 14 sec	59.13 sec
felt [35]	45 min 09 sec	17 min 15 sec	8 min 46 sec
Moon-Mooser [30]	37.74 sec	33.38 sec	24.88 sec
Moon-Mooser [36]	31 min 31 sec	17 min 14 sec	7 min 48 sec
Tarjan [4-4]	0.02 sec	0.27 sec	0.65 sec
Tarjan [6-6]	119 min 30 sec	14 min 27 sec	11 min 34 sec

Table 1
Runtime for computation of $CLIQUE3(G)$, SunFire 3800, 8 processors.

Because of the overhead parallel work is useful only for complex graphs either random with more than 80 nodes or with complex structure of edges as in Tarjan graphs (with at least 50 nodes), in felts and Moon-Moser graphs.

4.3 Parallelization of the Bron and Kerbosch algorithm

The Bron and Kerbosch algorithm (Figure 5) leads a depth first search over all combinations of a set K of nodes. Step by step K creates the clique, P contains all candidates for extension of K to a clique. D is the set of already considered nodes. If $|K| \geq k$ then K is the wanted clique. You can find more illustrations to this algorithm in [4].

This sequential algorithm works in polynomial time for a lot of graphs and it is better than other sequential algorithms. But for families of graphs with an exponential number of cliques (for instance felts, Moon-Moser graphs and Tarjan graphs), the computational time is exponential because Bron and Kerbosch enumerates all cliques. To reduce the time complexity in worst case we worked out a greedy variant of Bron and Kerbosch. For felts, Moon-Moser and Tarjan graphs we get a good computation time. This sequential algorithm tests only the first part of the tree of depth-first-search and enumerates not all cliques. However, it is possible, that the computed clique number k not comes true and greedy delivers a wrong result. After computing the first clique the

```
function clique(in G ∈ Graph,in k ∈ Nat) : Bool
let G = (V, E),
    function explore(in K, P, D ∈ Set of Vertices) : Bool
    begin
        if |K| ≥ k then return true fi;
        if P ≠ ∅ then
            choose v ∈ (P ∪ D) with deg(v) ≥ k − 1 and |P ∩ γ0(v)| maximal;
            if v ∈ P then
                P ← P − {v};
                if explore(K ∪ {v}, P ∩ γ0(v), D ∩ γ0(v)) then
                    return true
                fi;
                D ← D ∪ {v};
            fi;
            while P − γ0(v) ≠ ∅ do
                choose u ∈ (P − γ0(v)); P ← P − {u};
                if explore(K ∪ {u}, P ∩ γ0(u), D ∩ γ0(u)) then
                    return true
                fi;
                D ← D ∪ {u};
            od;
        fi;
        return false
    end explore
in
    if |V| ≥ k then
        return explore(∅, V, ∅)
    else
        return false
    fi;
end clique
```

Fig. 5. Sequential Bron and Kerbosch algorithm.

algorithm stops, but this is not necessarily a maximal clique. By means of the sequential Bron and Kerbosch algorithm we worked out an efficient parallel algorithm. In the sequential algorithm the computation time is different for the enumeration of the several cliques because of different numbers of recursive calls of `explore()`. The basic idea is to distribute the calls of the function `explore()` to different threads to get parallel computations. Load balancing works by using a task pool. One thread passes the recursive calls of `explore()` and replaces a call by a task, if for a given κ (for instance $\kappa = 0.75|V|$, experimental determined) $P \leq \kappa$. The other threads carry out the remaining calls of `explore()` to decide a clique. $\gamma_0(v)$ is the set of all neighbors of v, $deg(v)$ the degree of v. Additionally we can simultaneously choose more than one node

$v \in P \cap D$ and more than one node $u \in P \setminus \gamma_0(v)$.

The program fragment in Figure 6 shows the structure of our parallel implementation of the Bron and Kerbosch algorithm using a task pool.

```
public class BroKerPar implements Runnable   {
   NodeSet    p, d;         //   node sets P and D
      // ...
   public void run()  {     // checks number of recursions and
      // ...                   dependencies
      explore(p,d);
   }
   private void explore(NodeSet p,NodeSet d)  {   //...
   }
   // ...  several methods to work on the nodes
}
class BroKerParDivider extends BroKerPar   {  // thread to put
   // ...                              the tasks into the pool
   ThreadPool workPool;
   public void run()  {     // inits p with nodes
      // ...
      explore(p,d);
      try {workPool.waitForAll(true);}
      catch (InterruptedException e)   { //...
      }
   }
   public void explore(NodeSet p,NodeSet d)  {
      // here we decide whether to recurse or to create a
      // corresponding task for the pool:
      if (divideLimit*cl.g.numNodes < p_tmp.cardinality())
         explore(p_tmp,d.cut(conNodes));
      else   {   // ...
         workPool.addRequest(task);    }
   // ...
}
```

Fig. 6. Java code fragment: coarse structure of parallel Bron and Kerbosch algorithm.

In Table 2 are collected runtime results of the sequential and the thread parallel Bron and Kerbosch algorithm. The parallel implementation uses a task pool.

5 Conclusion

To decide $CLIQUE1(G, k)$ or rather to compute $CLIQUE2(G)$ or $CLIQUE3(G)$ it is useful to check if G has special qualities. For trees, per-

Graph [nodes]	sequential	parallel			
		$\kappa = 0.6\|V\|$	$\kappa = 0.75\|V\|$	$\kappa = 0.9\|V\|$	best speedup
random [60]	0.05 sec	0.01 sec	0.01 sec	0.01 sec	5.0
random [80]	17.97 sec	5.28 sec	4.90 sec	2.61 sec	6.9
felt [30]	58.49 sec	17.25 sec	15.51 sec	15.13 sec	3.9
felt [35]	9 min 27 sec	2 min 30 sec	1 min 59 sec	1 min 30 sec	6.3
M.-Mooser [30]	1 min 10 sec	18.86 sec	20.37 sec	18.75 sec	3.7
M.-Mooser [36]	18 min 48 sec	5 min 45 sec	5 min 30 sec	3 min 01 sec	6.2
Tarjan [4-4]	0.06 sec	0.02 sec	0.02 sec	0.02 sec	3.0
Tarjan [6-6]	3.61 sec	1.1 sec	0.86 sec	0.71 sec	5.1

Table 2
Runtime of the Bron and Kerbosch algorithm, CLIQUE3(G), task pool, 8 processors, SunFire 3800.

fect, chordal or bipartite graphs G or $\overline{G}$ exist good sequential polynomial time algorithms. In such a case we do not need parallelism. We can use a parallel computer with n processors (or n tasks) to test n qualities of G or $\overline{G}$ simultaneously in efficient time. In other cases it is helpful, if we use our developed and implemented sequential greedy Bron and Kerbosch algorithm. If, given G and k, the algorithm yields a k-clique then this result is true.

For general graphs the best computation time we get by use of our parallelized Bron and Kerbosch algorithm implemented in JAVA. We get an important speedup by the use of task pools. In practice we have to divide the whole task into parts - not too sparse, i.e., without a large overhead, and not too large, i.e., by exhausting all processors. We try to realize this in optimal manner with our implemented algorithms.

References

[1] H. Blaar, T. Rauber, and M. Legeler. Efficiency of thread-parallel java programs from scientific computing. In *IPDPS Workshop on Java for Parallel and Distributed Computing, IEEE*, 2002.

[2] A. Brandstädt, V. B. Lee, and J. P. Spinrad. *Graph Classes: A Survey.* SIAM Philadelphia, 1999.

[3] D. R. Butenhof. *Programming with POSIX Threads.* Addison-Wesley Longman Inc., 1997.

[4] P. Deussen and S. Kannapinn. *CLIQUE, KLÜNGEL, FILZ - Über die maschinelle Berechnung des Prädikats CLIQUE.* Technische Universität Berlin - FB Informatik, 1992.

[5] B. Eggers. Ordering on graphs and computations of clique. Technical Report 91-03, TU Berlin, FB 20, 1991.

[6] M. R. Garay and D. S. Johnson. *Computers and Intractability. A Guide*

to the Theory of NP-Completeness. W. H. Freeman and Company, New York, 1979.

[7] D. E. Lenoski and W.-D. Weber. *Scalable Shared-Memory Multiprocessing*. Morgan Kaufmann Publ., 1995.

[8] J. Mauro and R. McDougall. *Solaris internals: core kernel components*. Sun Microsystems Press, 2001.

[9] E. W. Mayr, H. J. Prömel, and A. Steger. *Lectures on Proof Verification and Approximation Algorithms*. Springer, 1998.

[10] S. Oaks and H. Wong. *Java Threads*. O'Reilly & Associates, Inc., 1999.

[11] T. Rauber and G. Rünger. *Parallele und verteilte Programmierung*. Springer, 2000.

[12] K. Simon. *Effiziente Algorithmen für perfekte Graphen*. Teubner, 1992.

[13] R. Vollmar and T. Worsch. *Modelle der Parallelverarbeitung*. Teubner, 1993.

[14] I. Wegener. *Theoretische Informatik*. Teubner, 1993.