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

Holger Blaar, Thomas Lange, Renate Winter
Martin-Luther-Universität Halle-Wittenberg
Institute für Informatik
06099 Halle (Saale), Germany
{holger.blaar,renate.winter}@informatik.uni-halle.de

Marcel Karnstedt
Technische Universität Ilmenau
Fakultät für Informatik und Automatisierung
Postfach 100 565, 98684 Ilmenau, Germany
marcel.karnstedt@tu-ilmenau.de

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.

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 [15]. 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 \( G \).

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 \( E \subseteq \{\{v_1, v_2\} : v_1, v_2 \in V \land v_1 \neq v_2\} \) of edges, \( |V| = n, k \in \mathbb{N}, k \leq n \).
Question: Does \( G \) have a \( k \)-clique, i.e., does exist a set \( V' \subseteq V \) with \( |V'| = k \), such that for every pair \( v_1, v_2 \in V' \) \( \{v_1, v_2\} \in E \) ?

Conclusion 1.2.2: \( CLIQUE1 \in PRAM–TIME(\Theta(n)) \) for \( \omega \geq id_N \) with identity function \( id_N(n) = n \).

With \( NP \) as the family of problems acceptable of a non-deterministic Turing machine with polynomial time complexity we get

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

If we have a sufficient number of processors the clique problem is efficiently parallelizable.
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], [14]), 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. [5]

2. Graphs to test parallel algorithms

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 parallel algorithms are practical. Test results of the sequential Bron and Kerbosch algorithm (see [3], [13]) with felt graphs, Moon-Moser graphs, and Tarjan graphs can be found in [5].

Definition 2.1 A felt graph $G_n = (V_n, E_n)$ of complexity $n$ contains $n$ components $S_1, S_2, \ldots, S_n$ with

1. every component $S_i$ is a circle with 5 nodes, $1 \leq i \leq n$,
2. $\forall i \forall j \forall u \forall v (1 \leq i \neq j \leq n \land u \in V_i \land v \in V_j \rightarrow (u, v) \in E_n)$.

Definition 2.2 A Moon-Moser graph $G_n = (V_n, E_n)$ of complexity $n$ contains $n$ components $S_1, S_2, \ldots, S_n$ with

1. every component $S_i$ has 3 isolated nodes, $1 \leq i \leq n$
2. $E_n = \{(u, v): u \in S_i \land v \in S_j \land i \neq j\}$.

Definition 2.3 A Tarjan graph $G_{m,n} = (V, E)$ contains $2n$ crowds $K^0_1, K^1_1, K^0_2, K^1_2, \ldots, K^0_n, K^1_n$, each with $m$ nodes. Two nodes $u \in V^x_i$ and $v \in V^y_j$ are not adjacent if $i = j$ and $x \neq y$.

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.

For graphs with an exponential number of cliques the Bron and Kerbosch algorithm does not work in polynomial time (see [5], [6]). Therefore our parallel algorithms were tested with graphs like felts, Moon-Moser, and Tarjan.

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., [11]. This is in contrast to most other programming languages where the operating system and a specific thread library like Pthreads [4] 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 or by using the interface Runnable. The method setPriority(int p) 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. An alternative synchronization mechanism is provided by the wait() and notify() methods of the Object class which also supports a list of waiting threads. 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., [11].

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 [9].

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 overhead 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 central or decentralized task queues.

For more details see, e.g., [12], [8]. The structure of our implementation of the used task pool is shown in Figure 4.

3.3. Runtime of multi-threaded programs

For measuring the runtime of a parallel application 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 operat-
// task pool implementation

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

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

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_{CI1} = \emptyset $ ($L_{CI1}$ 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 $L \setminus L_{CI1}$ a node $v$ with maximal degree.
   If $v_0 = ZERO$ store $v$, i.e., $v_0 = v$.
   $L_{CI1} = L_{CI1} \cup \{v\}$.
   If does not exist a node $v$ with degree $\geq k - 1$, then does not exist a $k$-clique in $G$. STOP
   If $|L_{CI1}| = k$ we have found a $k$-clique. STOP
   Else to step 4.
4. Compute $L$ as the set of all neighbors of $v$. If $L_{CI1} \setminus \{v\} \not\subseteq L$, go to the next step.
   Else to step 3.
5. $L = L_0 \setminus \{v_0\}, L_{CI1} = \emptyset$ 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 costs. 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 on a SunFire 3800.

<table>
<thead>
<tr>
<th>Graph [# nodes]</th>
<th>$p = 1$ parallel, $p = 8$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>without task pool</td>
</tr>
<tr>
<td>random [60]</td>
<td>13.16</td>
</tr>
<tr>
<td>felt [30]</td>
<td>87.00</td>
</tr>
<tr>
<td>felt [35]</td>
<td>2709.00</td>
</tr>
<tr>
<td>M.-Mooser [30]</td>
<td>37.74</td>
</tr>
<tr>
<td>M.-Mooser [36]</td>
<td>1891.00</td>
</tr>
<tr>
<td>Tarjan [4-4]</td>
<td>0.02</td>
</tr>
<tr>
<td>Tarjan [6-6]</td>
<td>7170.00</td>
</tr>
</tbody>
</table>

Table 1. Runtime (in seconds) for computation of $CLIQUE3(G)$, 8 processors.
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 \( |K| \geq k \) then \( K \) is the wanted clique.

function \texttt{clique}(in \( G \in \text{Graph.in} \ k \in \text{Nat} \) : \text{Bool}
let \( G = (V, E) \),
function \texttt{explore}(in \( K, P, D \in \text{Set of Vertices} \) : \text{Bool}
begin
  if \( |K| \geq k \) then return true fi;
  if \( P \neq \emptyset \) then
    choose \( v \in (P \cup D) \) with \( \text{deg}(v) \geq k - 1 \)
    and \( |P \cap \gamma_0(v)| \) maximal;
    if \( v \in P \) then
      \( P \leftarrow P - \{v\} \);
      if \( \text{explore}(K \cup \{v\}, P \cap \gamma_0(v), D \cap \gamma_0(v)) \) then return true fi;
      \( D \leftarrow D \cup \{v\} \);
    fi;
  while \( P - \gamma_0(v) \neq \emptyset \) do
    choose \( u \in (P - \gamma_0(v)) \);
    \( P \leftarrow P - \{u\} \);
    if \( \text{explore}(K \cup \{u\}, P \cap \gamma_0(u), D \cap \gamma_0(u)) \) then return true fi;
    \( D \leftarrow D \cup \{u\} \);
  od;
  return false
end \texttt{explore}
in
if \( |V| \geq k \) then
  \texttt{explore}(\emptyset, V, 0)
else
  return false
fi:
end \texttt{clique}

Figure 5. Bron and Kerbosch algorithm.

This sequential algorithm works in polynomial time for a lot of graphs. 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. We get a good computation time. 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 algorithm stops, but this is not necessarily a maximum clique. By means of the sequential Bron and Kerbosch algorithm we worked out an efficient parallel algorithm. The basic idea is to distribute the calls of the function \texttt{explore()} to different threads to get parallel computations. Load balancing works by using a task pool. One thread passes the recursive calls of \texttt{explore()} and replaces a call by a task, if for a given \( \kappa \) (for instance \( \kappa = 0.75|V| \), experimental determined) \( P \leq \kappa \). The other threads carry out the remaining calls of \texttt{explore()} to decide a clique. \( \gamma_0(v) \) is the set of all neighbors of \( v \), \( \text{deg}(v) \) 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.

<table>
<thead>
<tr>
<th>Graph [# nodes]</th>
<th>( p=1 )</th>
<th>( p=8 )</th>
<th>best speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \kappa )</td>
<td>( 0.6</td>
<td>V</td>
</tr>
<tr>
<td>random [60]</td>
<td>0.05</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>random [80]</td>
<td>17.97</td>
<td>5.28</td>
<td>4.90</td>
</tr>
<tr>
<td>felt [30]</td>
<td>58.49</td>
<td>17.25</td>
<td>15.51</td>
</tr>
<tr>
<td>felt [35]</td>
<td>567.00</td>
<td>150.00</td>
<td>119.00</td>
</tr>
<tr>
<td>M.-Mooser [30]</td>
<td>70.00</td>
<td>18.86</td>
<td>20.37</td>
</tr>
<tr>
<td>M.-Mooser [36]</td>
<td>1128.00</td>
<td>345.00</td>
<td>330.00</td>
</tr>
<tr>
<td>Tarjan [4-4]</td>
<td>0.06</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>Tarjan [6-6]</td>
<td>3.61</td>
<td>1.10</td>
<td>0.86</td>
</tr>
</tbody>
</table>

Table 2. Runtime (in seconds) and speedups \( S_p \) of the Bron and Kerbosch algorithm, \texttt{CLIQUET3}(G), task pool, \( p=8 \) processors, Sun-Fire 3800.

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 \texttt{CLIQUET1}(G, \( k \)) or rather to compute \texttt{CLIQUET2}(G) or \texttt{CLIQUET3}(G) it is useful to check if \( G \) has special qualities. For trees, perfect, chordal or bipartite graphs \( G \) or \( G \) exist good sequential polynomial time
public class BroKerPar implements Runnable {
    // node sets P and D
    NodeSet p, 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
    }
    // thread to put the tasks into the pool
    class BroKerParDivider extends BroKerPar {
        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); } // ...
        }
    }
}

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

algorithms. We can use a parallel computer with n processors (or n tasks) to test n qualities of G or 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