Exact Parallel Maximum Clique Algorithm for General and Protein Graphs

Matjaž Depolli,[1,‡] *Janez Konc,*[2,‡] *Kati Rozman,*[2] *Roman Trobec,*[1] *and Dušanka Janežič*[2,3*]

[1]Jožef Stefan Institute, Jamova 39, SI-1000 Ljubljana, Slovenia

[2]National Institute of Chemistry, Hajdrihova 19, SI-1000 Ljubljana, Slovenia

[3]University of Primorska, Faculty of Mathematics, Natural Sciences and Information Technologies, Glagoljaška 8, SI-6000 Koper, Slovenia

Keywords: protein comparison, graphs, maximum clique, parallel, algorithm

A new exact parallel maximum clique algorithm MaxCliquePara, which finds the maximum clique (the fully connected subgraph) in undirected general and protein graphs, is presented. First, a new branch and bound algorithm for finding a maximum clique on a single computer core, which builds on ideas presented in two published state of the art sequential algorithms is implemented. The new sequential MaxCliqueSeq algorithm is faster than the reference algorithms on both DIMACS benchmark graphs as well as on protein-derived product graphs used for protein structural comparisons. Next, the MaxCliqueSeq algorithm is parallelized by splitting the branch-and-bound search tree to multiple cores, resulting in MaxCliquePara algorithm. Ability to exploit all cores efficiently makes the new parallel MaxCliquePara algorithm markedly superior to other tested algorithms. On a 12-core computer the parallelization provides up to two orders of magnitude faster execution on the large DIMACS benchmark graphs and up to an order of magnitude faster execution on protein product graphs. The algorithms are freely accessible on *http://commsys.ijs.si/~matjaz/maxclique*.

* To whom correspondence should be addressed. E-mail: dusa@cmm.ki.si

‡These authors contributed equally.

Introduction

The maximum clique problem, one of the most challenging NP-complete problems[1] in computer science addresses important questions in bioinformatics[2, 3] and molecular modeling.[4-7] Molecules can be described as graphs with vertices representing atoms or group of atoms and with edges representing bonds. Detection of similarities between molecules, which implies finding an optimal alignment between their atoms or groups of atoms, can be expressed as a problem of finding a maximum clique in product graphs derived from the molecules that are being compared. Specifically, the maximum clique algorithm allows detection of protein structural similarities that are useful in protein classification or protein function prediction,[8] and searching large databases of chemical compounds, such as ZINC,[9] for structural similarities in small compounds,[4] which is a key to development of new drugs. RASCAL is an efficient clique-based algorithm for graph similarity calculation that has been applied to comparison of chemical graphs.[10-12] Using chemical heuristics, the algorithm is fine-tuned for comparison of small molecules.

A recent review,[13] identified several exact maximum clique algorithms, that were developed between 1990 and 2012.[14-19] However, all of these run on a single computer core and only two exact parallel maximum clique algorithms that could exploit computers with multiple cores exist.[20, 21] The algorithm of Pardalos *et al.*[20] is a master-slave type algorithm, in which a master process running on a dedicated core distributes jobs to slave processes running on the remaining cores that solve the maximum clique problem. The second algorithm, by McCreesh and Prosser,[21] is an algorithm adapted for use on large computer networks composed of up to 100 computers. Using Net File System for communication between processes, the algorithm reveals a significant overhead on the start-up, which makes it profitably applicable only to large and hard-to-solve instances of graphs, where that overhead can be compensated. The parallelization of maximum clique algorithms to exploit modern computer architecture with multiple cores is a severely under-explored field in computer science, although it has a high potential impact on molecular modeling.

Little is known regarding the efficiency of referred parallel maximum clique algorithms. A study of Pardalos' algorithm indicates that speedups between 1.8 and 2.6 can be achieved using a computer with four cores on two tested instances of DIMACS[a] graphs and on random graphs with up to 1000 vertices.[20] Another study of McCreesh and Prosser algorithm indicates that speedups of 10 and more can be obtained routinely on 25 or more desktop machines over large benchmark graphs that can require minutes or days to be solved on a single machine.[21] The literature however, appears to contain little, if any information on the effects of the parallelization on the speedup of maximum clique algorithms on modern computers with multiple cores. Likewise, little is known on the performance and on the comparison of such algorithms on a set of DIMACS graphs and protein-derived product graphs derived from real protein comparison applications, *e.g.*, performed by the ProBiS web server.[8, 22]

In this work we first develop a new exact sequential maximum clique algorithm MaxCliqueSeq, *i.e.*, one that is only able to exploit a single processor core. This algorithm runs faster on a single core than two currently leading maximum clique algorithms.[16, 23] Next, with our improved sequential algorithm taken as a starting point for the parallelization, we develop a new exact parallel maximum clique algorithm, MaxCliquePara, which balances its work and uses all the cores efficiently, by traversing multiple search tree branches at the same time. The new parallel algorithm is evaluated on DIMACS graphs and on a set of protein

[a] A publicly available collection of benchmark graphs for solving various graph-related problems, including the maximum clique problem, available from *http://dimacs.rutgers.edu*.

product graphs, which are target graphs for the newly developed algorithm. Our experimental results show that the MaxCliquePara algorithm is fast and effective on large protein graphs whose density, *i.e.*, the probability that an edge exists between two vertices, is very high, and that on average, it outperforms all state of the art maximum clique algorithms known to authors.

Methods

Graph Notation

An undirected graph $G = (V, E)$ consists of a set of vertices $V = \{v_1, v_2, ..., v_n\}$ and a set of edges $E \subseteq V \times V$ made up of pairs (u, v) of distinct vertices (Figure 1). Two vertices (u, v) are described as adjacent if $(u, v) \in E$. The neighborhood $\Gamma(v)$ of a vertex v is defined as $\Gamma(v) = \{u \in V \mid u \text{ is adjacent to } v\}$. The degree of a vertex v, denoted by $|\Gamma(v)|$, is the number of edges connected to v. A graph is complete if all its vertices are adjacent. A graph $H = (V_H, E_H)$, is a subgraph of G, if $V_H \subseteq V$ and $E_H \subseteq E$. A clique C is a complete subgraph of an undirected graph; the size of a clique C, denoted by $|C|$, is the number of its vertices. A maximum clique is the largest clique in a given graph. Coloring of a graph is a mapping that assigns one color to each vertex in such a way that no two adjacent vertices have the same color. The smallest possible number of colors with which a graph G can be colored is the chromatic number of G. If a graph G contains a maximum clique of size k, then at least k different colors are needed to color the vertices of this maximum clique. From this it follows that the size of the maximum clique that can be found in G is smaller or equal to the chromatic number of G.

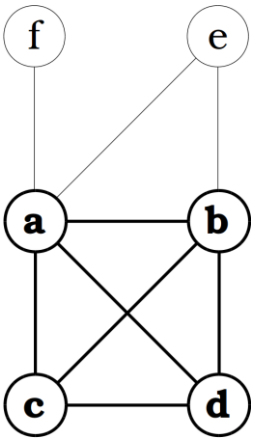

Figure 1. A general graph example. Vertices are circles, edges are lines. Maximum clique – set of vertices {a, b, c, d}, is highlighted in bold.

Protein Graphs

Proteins can be represented as protein graphs, as is schematically shown in Figure 2. In protein graphs, vertices have spatial coordinates, and vertices are placed at geometric centers of functional groups of protein surface amino acids. Vertices are labeled with five distinct colors corresponding to the five physicochemical properties, i.e., acceptor, donor, pi-pi stacking, aliphatic, and acceptor-donor, of the protein's surface amino acids at the resolution of functional groups. Two vertices u_i and u_j of a protein graph G are adjacent, that is, an edge $(u_i,u_j) \in E(G)$ exists between them, if the $distance(u_i, u_j) <$15 Å. To generate protein graphs, we followed the procedure[24] introduced by Konc and Janezic with an extension that we considered the entire protein surfaces as protein graphs, rather than only protein surface patches.

Protein Product Graphs

A pair of protein graphs can be compared by finding a maximum clique in their product graph, maximum clique being the superimposition that aligns the most vertices of the compared protein graphs (Figure 2). The protein product graph of two protein graphs G_1 and G_2 is defined on the vertex set $V(G_1, G_2) = V(G_1) \times V(G_2)$.[24] Each protein product graph vertex (u_i, v_i) is composed of two component vertices: a vertex from the first protein graph ($u_i \in G_1$), and a vertex from the second protein graph ($v_i \in G_2$). In general, a protein product graph has $x \times y$ vertices if the respective protein graphs have x and y vertices; however, we reduce its size by considering only vertices with identical component vertex colors (physicochemical properties). Additionally, component vertices must have similar neighborhoods in their corresponding protein graphs. The latter is determined as follows: the neighborhood of each protein graph vertex is defined as a sphere with diameter of 6 Å. Spatial dimensions are discretized into 24 intervals each with the length of 0.25 Å. A distance matrix representing discretized distances between all pairs of vertices in the neighborhood is calculated. The similarity of neighborhoods of the two component vertices p and r, which form a vertex of the protein product graph, is calculated using their corresponding distance matrices M_p and M_r as Similarity = $\sum_{1...m,1...n}(M_p^{ij} + M_r^{ij}) / \sum_{1...m,1...n}|M_p^{ij} - M_r^{ij}|$ where indexes i and j run over all m rows and n columns of each matrix (for more details see Ref. 24). The neighborhoods are considered similar if Similarity > 1.9.[24] Finally, two protein product graph vertices (u_i,v_i) and (u_j,v_j) are adjacent (an edge is inserted between them) if $(u_i,u_j) \in E(G_1)$ and $(v_i,v_j) \in E(G_2)$ and $|distance(u_i, u_j) - distance(v_i, v_j)| < 0.5$ Å, which means that the distances between respective first and second component vertices need to be almost the same in both protein graphs.

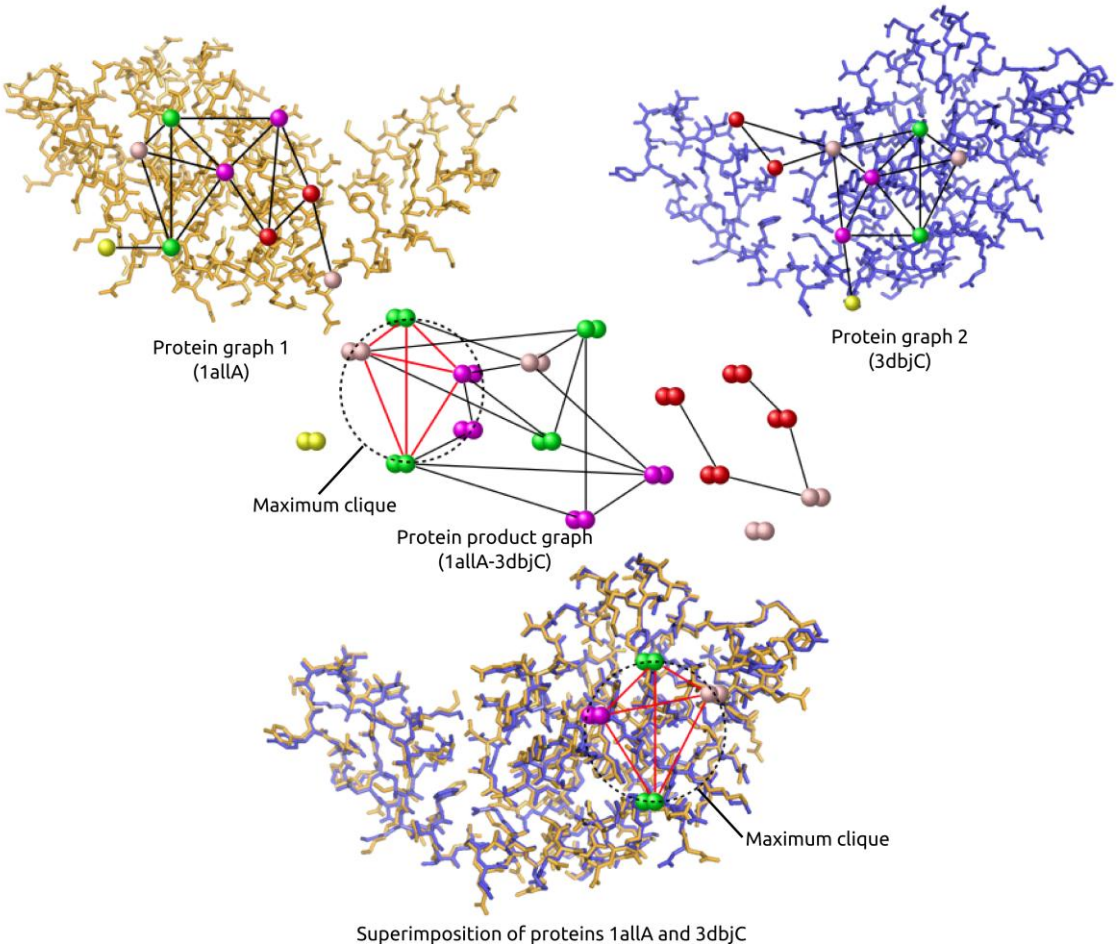

Figure 2. Protein structural comparison using MaxCliquePara maximum clique algorithm. Proteins are first converted to protein graphs (top) - for clarity, only a section of each protein graph is shown. Vertices are colored according to their physicochemical properties, *i.e.*, acceptor (red), donor (green), pi-pi stacking (pink), and aliphatic (yellow). A protein product graph is constructed from both protein graphs (center). A maximum clique of four vertices connected with red edges indicates the similarity between proteins. Finally, the two compared proteins are superimposed (bottom) according to the best alignment of vertices represented by the maximum clique.

Accordingly, we constructed ten benchmark protein product graphs, each from a pair of protein structures from the Protein Data Bank (PDB).[25] The name of each protein product graph, *e.g.*, *1allA-3dbjC* reflects the PDB and Chain IDs of the compared proteins. To measure the effect of protein size, *i.e.*, number of amino acids, on the performance of the maximum clique search, we constructed product graphs from proteins with ~50 to ~2000 amino acids. In addition, we considered protein pairs that share ~10-95 % sequence identity. The resulting protein product graphs serve as a sample of typical inputs for the proposed algorithm, and we envision their use as standard future tests for newly developed maximum clique algorithms.

Generic Maximum Clique Algorithm

A generic sequential maximum clique algorithm employing a branch-and-bound approach is shown in Figure 3.

```
1.  procedure Generic (U, C)
2.  while (U not empty)
3.        <v, n> = <vertex, number> from U where number is maximal
4.        U = U \ {<v, n>}
5.        if (|C| + n > |Cmax|)
6.              C = C ∪ {v}
7.              U1 = Color (U ∩ Γ(v))
8.              if (U1 not empty)
9.                    Generic (U1, C)
10.             else if (|C| > |Cmax|)
11.                   Cmax = C
12.             end if
13.             C = C \ {v}
14.       end if
15. end while
```

Figure 3. Generic maximum clique algorithm. Backslash in $X \setminus Y$ is the relative complement (also the set difference) between sets X and Y. *Input: U*, a set of pairs *<vertex, number>*, where *vertex* is a graph vertex, and *number* is its degree; empty sets C and C_{max}, which will hold vertices of the current and maximum clique, respectively. *Execution:* on each step, a vertex with maximum number is taken from U; if bounding condition is true, the vertex is added to current clique in C; coloring replaces numbers in U with colors (see below) in U_1, followed by a recursive call to function Generic; after recursion, if $|C| > |C_{max}|$, the current clique becomes the maximum clique. *Output:* C_{max} holds vertices of the maximum clique.

Approximate Graph Coloring Algorithm

A coloring algorithm assigns colors (or numbers) to vertices of a graph, so that no two adjacent vertices have the same colors (or numbers). The number of colors assigned by a coloring algorithm is used in the maximum clique algorithm as the bounding condition (see section *Graph Notation*). Although vertex coloring is an NP-complete problem,[1] fast approximate algorithms[26] exist that solve it based on the "greedy strategy", with a complexity of $O(n^2)$ with respect to the number of vertices n to be colored. An example of approximate coloring of a graph from Figure 1 is shown in Figure 4.

At the first step of this algorithm, the set of graph vertices V_{copy} is empty, therefore $V_{copy} = V$. In the next step, vertex a is selected as the vertex with the highest degree from V_{copy}; it is colored using the first available color, *i.e.*, color no. 1 - red (colors are represented as natural numbers), and added to the set of colored vertices U. The neighborhood vertices of vertex a, marked with thick edges in the graph and gray color in V_{copy} (all vertices), are then removed from V_{copy}, and vertex a is removed from V. Since V_{copy} becomes empty, it is refilled from V, and the current color is changed to 2 - green (color number is increased by one). Next, vertex b is selected from V_{copy}, colored green, and added to the set U. The neighborhood vertices of vertex b, marked with gray color in V_{copy} (vertices c, d, and e), are removed from V_{copy}, and vertex b is removed from V. The procedure repeats itself until V is empty, when the graph is colored and the algorithm stops.

The number of colors used to color the graph, *i.e.*, in this example four colors were used, presents the upper bound to the size of the maximum clique that can be found in this graph, and the maximum clique algorithm uses this value to prune the branches of the search tree.[27]

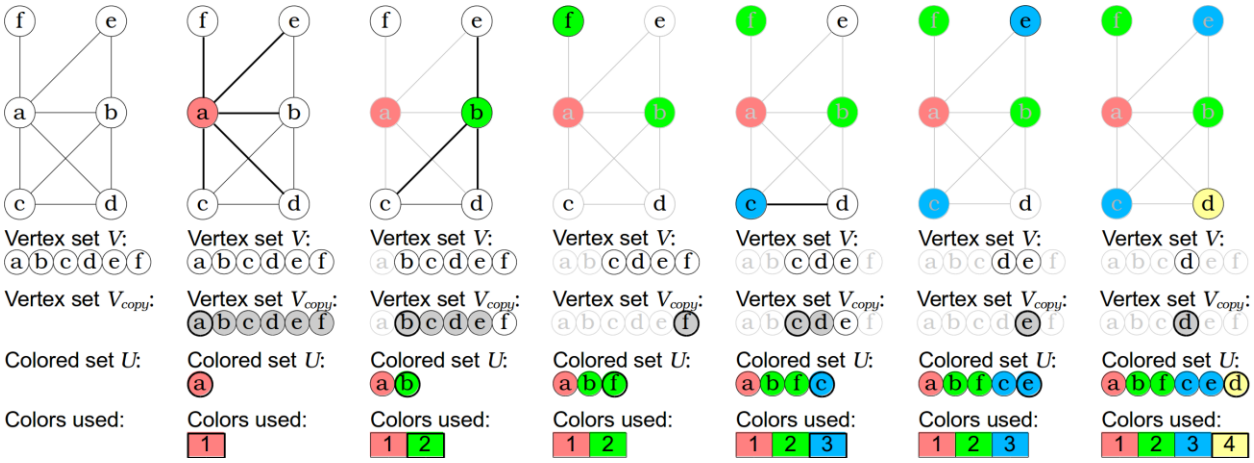

Figure 4. Approximate coloring algorithm trace. *Input*: leftmost panel graph (same as in Figure 1). *Execution*: the algorithm proceeds from left to right and in each consecutive panel it colors one vertex of a graph. *Output*: graph colored with 4 colors, and a working set of colored vertices *U*, ordered by their colors (or numbers) (rightmost panel).

In the MaxCliqueDyn algorithm,[16] an approximate coloring algorithm was introduced that was later adopted by the MCS algorithm[15] and enhanced by addition of bit-strings in the BBMC algorithm.[23] All three algorithms follow the same realization that color-based vertex ordering is only needed above a threshold, which is calculated as $k_{min} = |C_{max}| - |C| + 1$, where $|C_{max}|$ is the size of the current maximum clique and $|C|$ is the size of the clique found on the current branch of the search tree. Vertices with colors below k_{min} can remain in their original order, as they will never be used in further recursions as a root of a sub-branch. In this way, the algorithm keeps most of the vertices in the working set U in a non-increasing degree ordering. This results in graphs colored with fewer colors and, consequently, more tightly bound to the size of the maximum clique. In comparison to previous coloring algorithms[28] our approximate coloring algorithm[16] consistently reduces the number of steps needed to find a maximum clique and consequently lessens the time required to find a maximum clique.

Initial Vertex Ordering

Initial order in which vertices are presented to the approximate coloring algorithm significantly affects the performance of the maximum clique algorithm.[29] The coloring is tighter, *i.e.*, fewer colors are needed, if the vertices presented to the approximate coloring procedure are ordered by non-increasing degree,[16] so that, if $|\Gamma(u)| \geq |\Gamma(v)|$, then u is placed before v. However, some freedom remains in the ordering of vertices having the same degrees. Tomita *et al.*[15] described an efficient initial ordering of vertices of the same degree with the introduction of *ex-degree*, defined for a given vertex as the sum of degrees of all its adjacent vertices. In preprocessing, *ex-degree* is calculated, followed by the initial sorting by non-increasing degree and non-increasing *ex-degree* for vertices of equal degrees. After the vertices are sorted, they are renumbered, so that the vertex with the highest degree has index one, the one with second highest degree has index two, and so on. The adjacency matrix,[30] *i.e.*, an $n \times n$ matrix for a graph with n vertices in which element a_{ij} is non-zero if edge exists between vertices i and j, is then reconstructed with the renumbered vertices. The consequence of renumbering is a better localization of memory access and therefore more efficient use of the cache memory. Additionally, an adjunct vertex set (set V in Figures 4 and 5) that maintains the initial order of vertices throughout the execution of the maximum clique algorithm is introduced.

Use of Bit-strings

Recently, Segundo *et al.* introduced a maximum clique algorithm[23] that uses the so called bit-strings (also called bit-boards) to encode adjacency matrix and adjunct vertex set V. Bit-strings offer extremely fast set operations such as "union", "intersection", and "complement", but are slower than lists in counting the number of elements, or in extracting the elements with the lowest or the highest number. These deficiencies however are alleviated by modern processors encompassing fast instructions for bit manipulation. In addition, bit-strings are more cache-efficient for storing sets than the traditionally used arrays, because they store up to eight elements per byte in contrast to arrays that store at most one element per byte. On the other hand, the order of elements in a bit-string is predefined and cannot be changed, as in adjacency matrix and adjunct vertex set. Ordered sequences of vertices, such as the vertex sequence U, the output of coloring algorithm and used for branching, are stored in arrays.

MaxCliqueSeq Maximum Clique Algorithm

We develop a new maximum clique algorithm, MaxCliqueSeq (Figure 5), by combining the existing algorithmic building blocks. The algorithm is composed of approximate coloring algorithm,[16] initial vertex ordering algorithm,[15] and uses bit-strings[23] to encode the adjunct set V and the adjacency matrix; adjacency matrix is not shown in Figure 5, since it is used internally in function Γ to determine the neighborhood of a vertex. These algorithms and concepts were described in the previous sections. This combination of existing algorithms,[15, 16, 23] building blocks of different maximum clique algorithms, has proven the fastest on most DIMACS and protein product graphs out of different combinations tested. For example, an experimental algorithm that used a more advanced coloring algorithm,[15] performed worse in our test.

In the implementation, we avoid copying of variables by value; we also keep the code compliant with cache. Consequently, C, C_{max}, and the adjacency matrix, are global variables. Instead of performing the complement of the neighborhood $\Gamma(v)$ at each step in the coloring function (see line 47 in Figure 5), a complemented copy of the adjacency matrix is prepared in the preprocessing step. The coloring function then uses the complemented adjacency matrix instead of the original adjacency matrix, decreasing the number of operations required. The source code, written in C++ standard 2011, can be compiled with GCC 4.6 and should be portable to any other modern compiler, since it employs only standard C++ functions and the standard template library.

The trace of this algorithm on an example graph from Figure 1 is shown in Figure 6. The algorithm traverses the search tree from left to right, following the numbered arrows. It starts by adding vertex a to the current clique C (step 1). Since vertex b is connected with vertex a, the clique C is then extended by vertex b (step 2); at the same time, vertex f, which is not connected to vertex b, is removed from U, and can never be considered again within this branch (steps 3 to 5).

```
1.  procedure MaximumCliqueSeq
2.   Cmax = {}   // global variable
3.   C = {}
4.   U = Order (V)
5.  renumber vertices in U, V and adjacency matrix
6.   Expand (V, U, C)
7.        renumber vertices in C back to their original numbering
8.
9.  procedure Expand (V, U, C)
10.       while (|U| > 0)
11.            <v, n> = first element of U
12.            V = V \ {v}
13.            if (|C| + n > |Cmax|)
14.                 if (|V ∩ Γ(v)| > 0)
15.                      Expand (V∩ Γ(v), Color (V, |Cmax| - |C|), C ∪ {v})
16.                 else if (|C| ≥ |Cmax|)
17.                      Cmax = C ∪ {v}
18.                 end if
19.            end if
20.       end while
21.
22. function Order (V)
23.  R = V
24.       U = {}
25.            // define Rmin as a subset of vertices with minimum degree in R
26.       while (|Rmin| ≠ |R|)
27.       select p as a vertex with minimal ex-degree in Rmin
28.       R = R \ {p}
29.        U[|R|].vertex = p
30.       degree (q) = degree (q) – 1, where (q ∈ R) & (q ∈ Γ(p))
31.  end while
32.  N = Color (Rmin, 0)
33.  U[0..|N|-1] = N
34.  Dmax = maximum degree (v), where v ∈ V
35.  for (i = |N| to |U|-1)
36.       U[i].number = min (U[i-1]+1, Dmax+1)
37.  end for
38.
39. function Color (V, kMin)
40.  U = {}
41.  k = 1
42.  while (|V| > 0)
43.       Vcopy = V
44.       while (|Vcopy| > 0)
45.            v = first element of Vcopy
46.            V = V \ {v}
47.            Vcopy = Vcopy \ Γ(v)
48.            if (k > kmin)
49.                 add pair <v, k> to U
50.            end if
51.       end while
52.       k = k + 1
53.  end while
54.  return U
```

Figure 5. The MaxCliqueSeq algorithm. Overscore in $\overline{X}$ is complement of a set X and backslash in $X \setminus Y$ is the relative complement (also the set difference) between sets X and Y. *Input*: V, a set of graph vertices; U, an ordered sequence of pairs *<vertex, number>*; adjacency matrix; C and C_{max}, current and maximum clique, respectively (empty at start). *Output*: C_{max} holds vertices of the maximum clique.

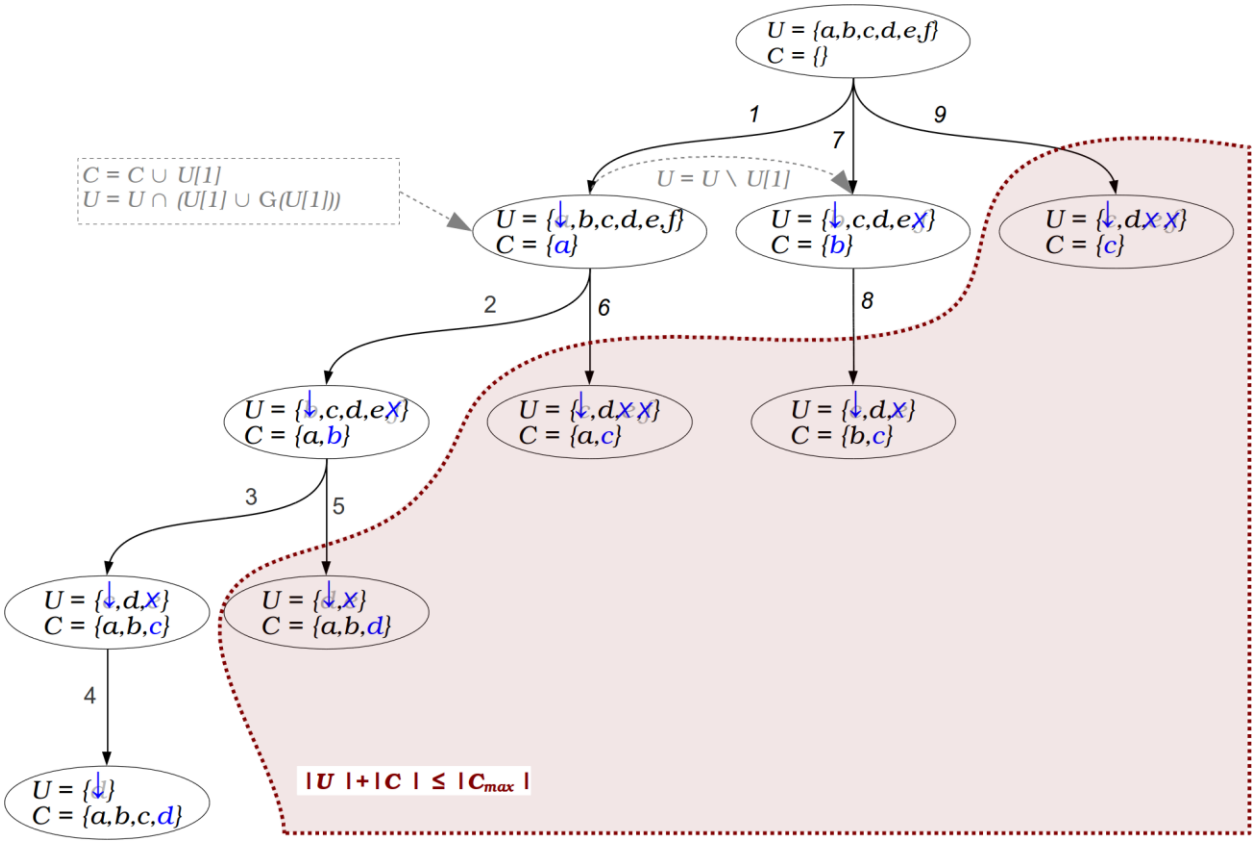

Figure 6. MaxCliqueSeq maximum clique algorithm trace. In each step, first vertex is removed from the working set U (blue arrow ↓ in oval), and added to the current clique C (blue vertex in oval); vertices not connected to the first vertex are removed from U (blue cross × in oval). Each oval is one step of the algorithm and recursive calls are arrows numbered 1–9 to indicate their sequence. Branches, not executed due to bounding condition $|U| + |C| \leq C_{max}$ are in the reddened area enclosed by a dashed line. Vertex coloring is omitted for clarity; $|U|$ is used in the bounding condition instead of the number of colors. In step 4, a maximum clique C_{max}={a,b,c,d} is found.

MaxCliquePara Maximum Clique Algorithm

Here, we introduce the new parallel maximum clique algorithm, MaxCliquePara, based on the MaxCliqueSeq algorithm outlined in the previous section. A flowchart of the MaxCliquePara algorithm is shown in Figure 7. The algorithm employs the multi-threading techniques, which are supported in most programming languages without extra libraries or other sort of software support. This makes the algorithm portable to other languages and operating systems. It can run on most modern multi-core computer architectures. The parallel algorithm behaves identically to the MaxCliqueSeq algorithm when executed on a single core, but takes slightly longer to execute, due to the added thread management code. However, when executed on multiple cores it easily compensates this overhead if the maximum clique problem to be solved is not trivial, *e.g.*, high density graphs.

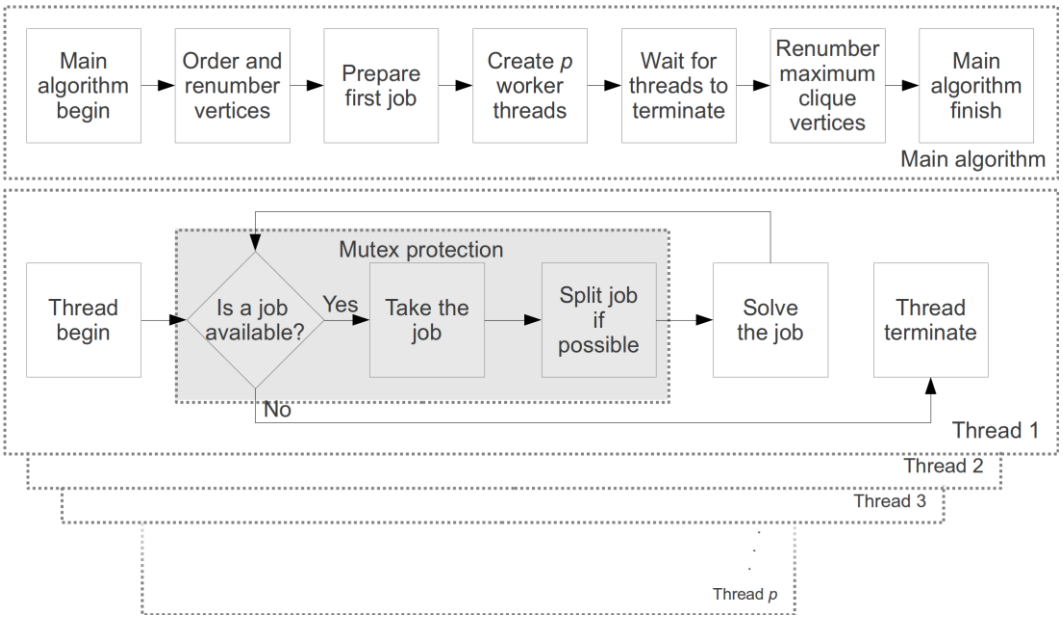

Figure 7. Flowchart of the MaxCliquePara parallel maximum clique algorithm, consisting of main algorithm (top) and parallel thread execution (bottom).

Main Algorithm

The main algorithm (see Main algorithm in Figure 7) starts with a preprocessing step consisting of the initial ordering and renumbering of vertices. This step is not parallelized, since its execution time, in comparison to the total time needed to find a maximum clique, is significant only on small and easily solvable graphs. These graphs are not the target graphs of our algorithm and can be often searched more efficiently by a sequential algorithm.

The next step, which also executes sequentially, is the preparation of the first job. This step generates a data structure comprised of an empty current clique C and maximum clique C_{max}, a color-numbered working set of vertices U, and an adjunct set of vertices V.

The algorithm proceeds with the creation of p worker threads, the number of which can be set at runtime. Setting p to the number of cores has proven as the most efficient in our tests. Worker threads are created and started in a sequence, each thread on its own core. At creation, threads may also be assigned affinity to be associated with a single core throughout their lifetime. The alternative is to leave the thread management completely to the operating system, which may rotate their host cores, causing a lower cache hit rate. Our preliminary tests, however, failed to reveal any significant difference between run times of the algorithm with or without affinity settings.

The final step of the main algorithm is to wait for all the threads to terminate and to free their resources. The maximum clique found is stored in the shared variable C_{max}, and is renumbered back to the original numbering, before being returned to the user.

Parallel Thread Execution

Each newly created thread (see Threads in Figure 7) enters a loop, and first checks for available jobs. If no jobs are available, all the work has already been assigned, thus thread terminates. If a job is available, the thread acquires it and attempts to split it into two jobs: if the number of vertices in the job's working set U is above the user–defined threshold, the first vertex is removed from U and the modified working set is used to create a new job. The threshold size of U for splitting the job, $|U| >= 5$, was determined with preliminary tests, as splitting jobs with $|U| < 5$ slowed down the execution. The new job can then be acquired by

a new thread, when a core becomes available. The thread then proceeds with a search on the modified working set with the removed vertex added to its current queue. If the job cannot be split as defined then the thread processes the whole job and the next thread in line will find no available jobs.

The final and most time demanding activity is the job solution, *i.e.*, searching for the maximum clique in the working set of vertices U of the acquired job. This is done in function Expand (Figure 5), whose working set of vertices U is defined by the current job rather than being the result of preprocessing as in the sequential algorithm.

Communication between threads is achieved through use of shared variables, *e.g.*, C_{max}, which are protected from multiple concurrent modifications and accesses with mutual exclusion or mutex (see Figure 7).[31] Since the use of mutexes exacts an overhead, their number is kept to a minimum. Shared variables, *e.g.*, integers, which can be read and modified, using atomic operations, are allowed for unprotected access without use of mutexes. For example, the global variable C_{max} is write-protected by a mutex, whereas $|C_{max}|$, which is an integer, is not. The C_{max} could be a thread-private variable and only synchronized upon completion, which could result in faster execution. However, our preliminary tests showed that such an optimization of the storage and protection policy of C_{max} would have almost no effect on the execution time and would also complicate the program.

Threads explore multiple disjunct branches of the search tree, each branch rooted in a different starting vertex as shown in Figure 8 on an example graph from Figure 1. Thread 1 explores a branch that has vertex a as a root and all other vertices as candidates. Thread 2 explores another branch, obtained by first job splitting, and has vertex b as a root and all other vertices, except vertex a, as candidates. The number of candidate vertices in the working set U decreases with each additional thread, and the average execution time decreases accordingly, thus the complex, most time-consuming branches, are searched first. When the threads exploring complex branches finish their work, the only remaining threads are those that explore simple branches, which allows for low idle time and good load-balancing.

In contrast to the sequential algorithm (Figure 6), the parallel algorithm explores two branches at the same time (Figure 8). This decreases number of steps, i.e., 7 as opposed to 9 in the sequential algorithm, and results in faster execution time (speedup) of the MaxCliquePara algorithm. However, a branch can remain unpruned (see blue shaded oval in Figure 8) in parallel algorithm, whereas the same branch is pruned in the sequential algorithm (Figure 6). This branch is not pruned, because at the time Thread 2 starts traversing this branch, Thread 1 did not find the maximum clique with four vertices yet (as is the case in the sequential algorithm), and thus Thread 2 only has the maximum clique with three vertices in the bounding condition. Therefore, the blue shaded branch searched by Thread 2 is not pruned in the parallel algorithm, which makes Thread 2 to traverse one more branch compared to the sequential algorithm trace. This inefficiency of the parallel algorithm is compensated by the concurrent execution of three threads compared to a single thread in the sequential algorithm. The inverse example could be envisioned, in which a maximum clique found by one thread would help prune branches of another thread, thus acting advantageously on the pruning process in the parallel algorithm. Judging from the super-linear speedups achieved on general and protein graphs (see Results), this is often the case.

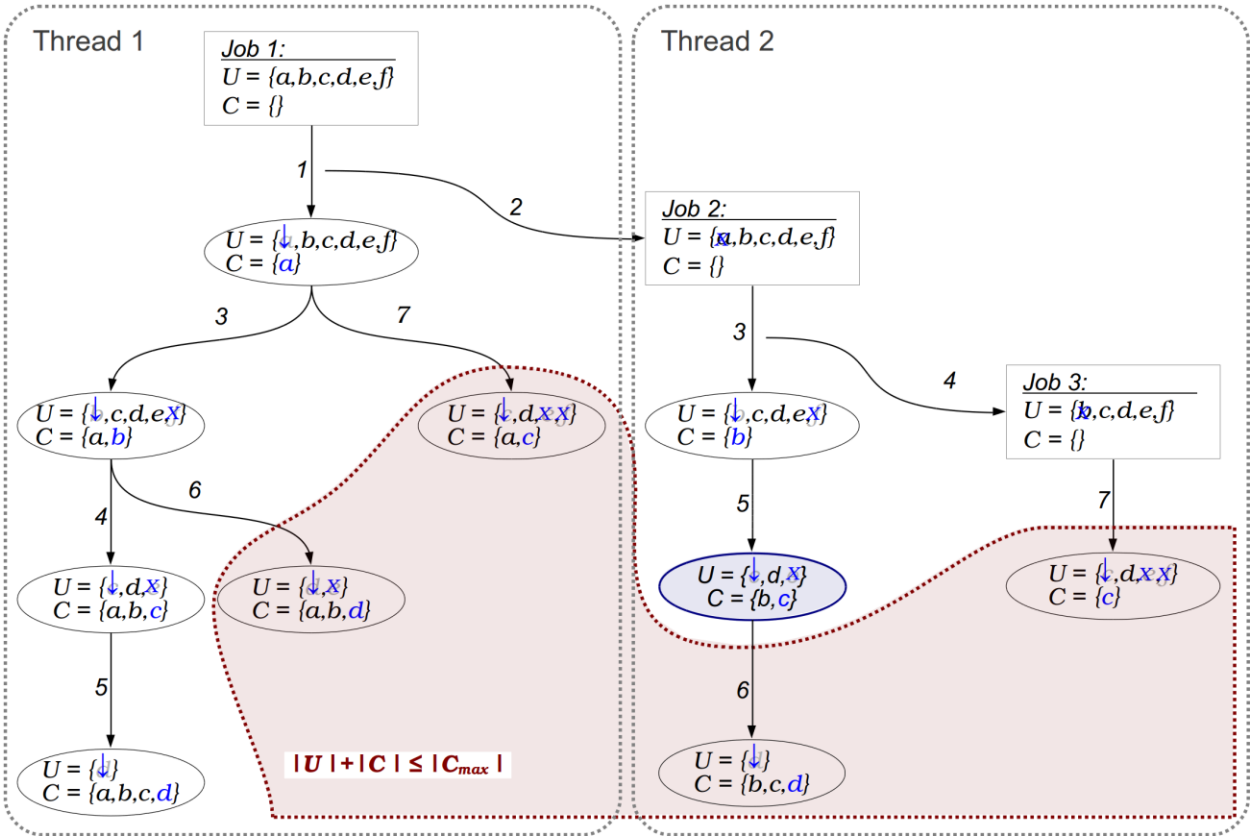

Figure 8. Parallel MaxCliquePara algorithm trace executed as two threads (Thread 1 and Thread 2). Each thread execution is the same as described in Figure 6 except that in Thread 2, a new step 5 (blue shaded oval) represents branch that was pruned by MaxCliqueSeq, but not by MaxCliquePara. In step 5 of Thread 1 a maximum clique C_{max}= {a,b,c,d} is found.

Results

We evaluated the performance of the MaxCliquePara algorithm in terms of execution times and speedups on DIMACS graphs, used in standard benchmarking of maximum clique algorithms, and on typical protein product benchmark graphs to determine the algorithm's performance on general and protein graphs. Our algorithm was compared with MaxCliqueDyn[16] and BBMC[23] algorithms, which are currently the fastest maximum clique algorithms available, and Bron-Kerbosch,[32] which is among the most widely used maximal clique algorithms. All experiments were repeated 15 times with negligible standard deviations in execution times, therefore average results are shown. The test computer with a dual CPU 2.30 GHz Intel Xeon E5-2630, each with six physical cores, running server version of Ubuntu 12.04 was used.

General DIMACS Graphs

Results on a single core show that on most general DIMACS graphs, MaxCliqueSeq is faster than the BBMC and MaxCliqueDyn algorithms, whereas the Bron-Kerbosch algorithm is always the slowest (Table 1).

Table 1. Execution times for DIMACS graphs[a] on a single core using three exact maximum clique algorithms, with results given as a mean of 15 trials ± standard deviation

	Execution time[b] [s]				
Graph name	MaxCliquePara	MaxCliqueDyn	BBMC[c]		Bron-Kerbosch
hamming8-2	**0.000865 ± 8.5e-05**	0.00375 ± 4.8e-05	0.00135 ± 7.9e-05	(N/A)	> 2h
brock200 2	0.00391 ± 0.00072	0.00735 ± 0.00012	**0.00369 ± 0.00021**	(<0.001)	0.18 ± 0.0052
keller4	**0.00868 ± 0.00073**	0.014 ± 0.00023	0.0111 ± 0.00063	(<0.001)	1.25 ± 0.0038
san400 0.5 1	0.00966 ± 0.00053	**0.00732 ± 7.4e-05**	0.0103 ± 5e-05	(0.016)	2990 ± 3.5
p hat300-2	**0.00985 ± 5.9e-05**	0.0212 ± 0.00021	0.0112 ± 0.00019	(<0.001)	24 ± 0.048
p hat500-1	0.0129 ± 6.7e-05	0.0181 ± 0.00027	**0.0122 ± 2.4e-05**	(<0.001)	0.228 ± 0.0006
brock200 3	**0.0139 ± 0.0014**	0.0294 ± 0.00029	0.0151 ± 0.00082	(<0.001)	1.93 ± 0.039
hamming10-2	**0.0165 ± 7.8e-05**	4.54 ± 0.0078	0.0444 ± 6.9e-05	(0.063)	> 2h
san200 0.9 2	**0.0268 ± 0.00025**	0.371 ± 0.0015	0.134 ± 0.0016	(0.062)	> 2h
san200 0.9 3	**0.0299 ± 0.0025**	1.89 ± 0.013	1.82 ± 0.031	(0.015)	> 2h
C125.9	**0.0316 ± 0.0018**	0.06 ± 0.0038	0.0413 ± 0.00065	(0.016)	1690 ± 1.7
p hat700-1	**0.045 ± 0.0038**	0.0677 ± 0.0039	0.0451 ± 0.0025	(0.047)	1.08 ± 0.0045
hamming8-4	0.0465 ± 0.0045	**0.0367 ± 0.00021**	0.0468 ± 0.0027	(0.015)	11.3 ± 0.032
brock200 4	**0.048 ± 0.0004**	0.0961 ± 0.0013	0.0526 ± 0.0007	(0.063)	7.91 ± 0.031
san400 0.7 2	**0.0661 ± 0.00038**	0.0944 ± 0.00029	0.297 ± 0.0014	(0.063)	> 2h
johnson16-2-4	**0.094 ± 0.0082**	0.225 ± 0.0029	0.104 ± 0.0068	(0.062)	1.18 ± 0.01
sanr200 0.7	**0.118 ± 0.0094**	0.218 ± 0.0027	0.154 ± 0.0048	(0.125)	27.1 ± 0.077
san400 0.9 1	**0.122 ± 0.0012**	21 ± 0.047	4.47 ± 0.014	(0.031)	> 2h
san200 0.9 1	0.145 ± 0.012	**0.0547 ± 0.00023**	0.175 ± 0.0074	(0.094)	> 2h
gen200 p0.9 44	**0.25 ± 0.0078**	0.956 ± 0.0037	0.596 ± 0.0056	(0.187)	> 2h
p hat1000-1	**0.263 ± 0.0044**	0.35 ± 0.0038	0.301 ± 0.00067	(0.421)	5.57 ± 0.022
MANN a27	**0.265 ± 0.0064**	1.99 ± 0.025	0.316 ± 0.004	(0.187)	> 2h
sanr400 0.5	**0.285 ± 0.0064**	0.526 ± 0.0047	0.311 ± 0.019	(0.327)	12.6 ± 0.037
brock200 1	**0.287 ± 0.0086**	0.538 ± 0.0062	0.389 ± 0.0054	(0.312)	167 ± 0.45
p hat500-2	**0.303 ± 0.0068**	0.852 ± 0.0051	0.487 ± 0.0018	(0.39)	> 2h
san400 0.7 1	0.362 ± 0.0072	**0.217 ± 0.00069**	0.438 ± 0.0014	(0.125)	> 2h
gen200 p0.9 55	0.47 ± 0.01	**0.469 ± 0.0015**	0.543 ± 0.011	(0.437)	> 2h
san400 0.7 3	**0.564 ± 0.0025**	1.43 ± 0.0063	1.5 ± 0.0097	(0.437)	> 2h
p hat300-3	**0.993 ± 0.0091**	2.51 ± 0.013	1.66 ± 0.012	(1.31)	> 2h
san1000	1.7 ± 0.0095	**0.284 ± 0.0019**	1.75 ± 0.009	(0.375)	> 2h
p hat1500-1	**2.38 ± 0.0093**	2.93 ± 0.017	2.92 ± 0.0029	(3.92)	> 2h
p hat700-2	**2.38 ± 0.0081**	7.06 ± 0.02	3.68 ± 0.0024	(3.79)	> 2h
sanr200 0.9	**13.6 ± 0.077**	25.5 ± 0.065	30.2 ± 0.35	(13.9)	> 2h
p hat500-3	**55.7 ± 0.22**	183 ± 0.6	105 ± 0.57	(76.1)	> 2h
sanr400 0.7	**70.2 ± 0.29**	99.4 ± 0.25	94.4 ± 0.47	(102)	> 2h
brock400 4	**94.6 ± 0.25**	154 ± 0.81	145 ± 0.6	(143)	> 2h
p hat1000-2	**116 ± 0.67**	230 ± 1.3	200 ± 0.86	(193)	> 2h
brock400 2	**117 ± 0.77**	171 ± 1	161 ± 0.87	(140)	> 2h
MANN a45	**119 ± 1.1**	1395 ± 19	156 ± 0.28	(42.4)	> 2h
brock400 3	**196 ± 4.5**	348 ± 2.9	332 ± 2	(240)	> 2h
brock400 1	**279 ± 6.5**	368 ± 2.4	377 ± 0.81	(348)	> 2h
p hat700-3	**1118 ± 2**	3208 ± 40	2080 ± 14	(1640)	> 2h
C250.9	**1299 ± 14**	1802 ± 21	2256 ± 33	(1290)	> 2h

[a] Statistics of the DIMACS graphs are given in Supporting Information in Table ST1.

[b] Graphs are sorted by increasing execution times; fastest execution time in each row is in bold.

[c] Execution times achieved by our implementation of the BBMC algorithm and times reported by Segundo *et al.*[18] (in parentheses) are given. Latter times cannot be compared with our results due to no reproducibility. See Supporting Information.

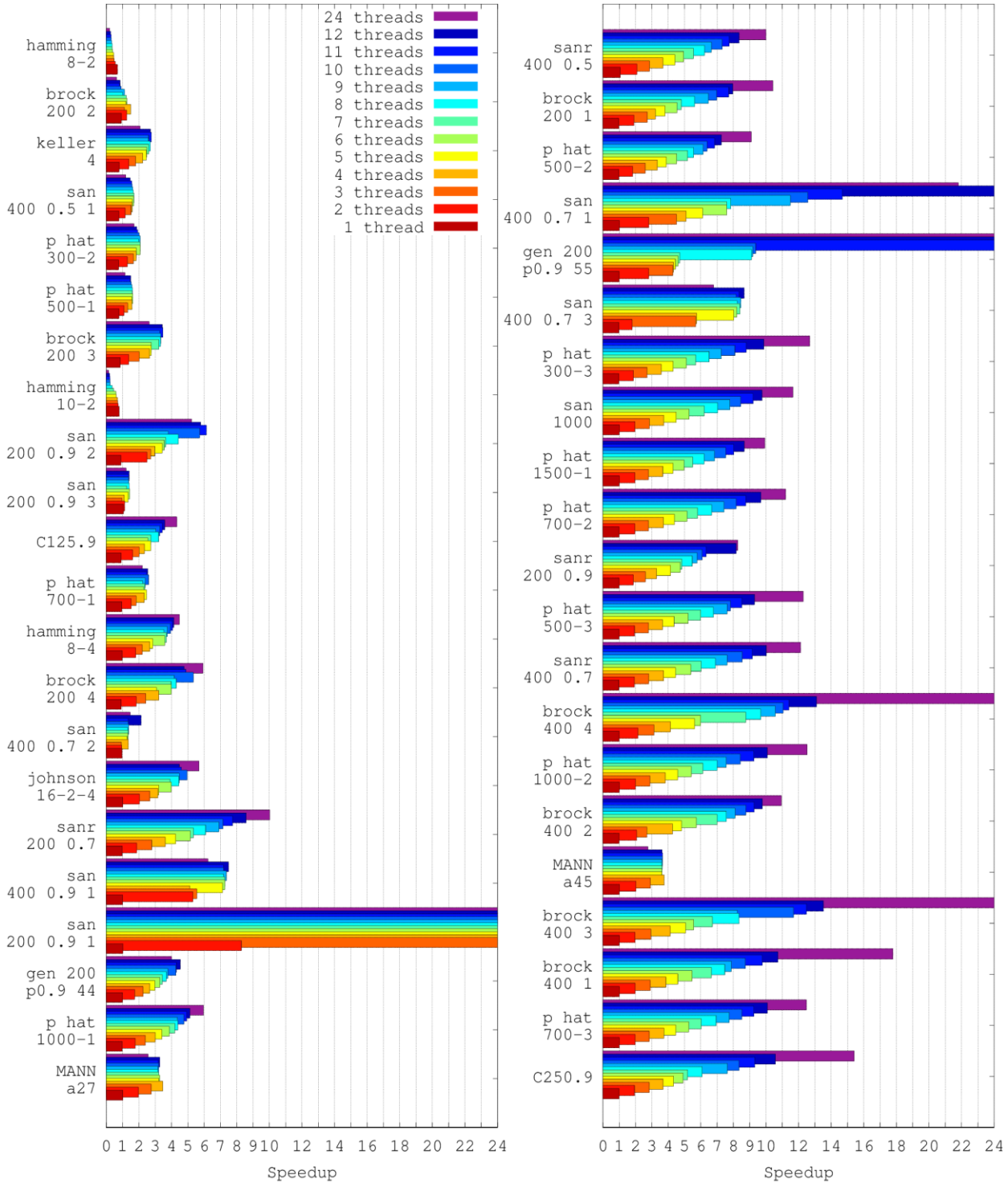

Figure 9. Speedup of the parallel MaxCliquePara algorithm on DIMACS graphs. Graphs are sorted by increasing execution times on a single core. Exact values for speedups are given in Supporting Information in Table ST2.

Speedups of the MaxCliquePara algorithm on multiple cores are shown in Figure 9. Speedups differ greatly between the different DIMACS graphs and there are several super-linear[b] speedups, which occur mostly in large and dense instances of *san* and *brock* graph families (Figure 9). There were 8 super-linear speedups out of 43 graphs tested when executing on two cores (see Table ST2). Super-linear speedup occurred, for example, in *san200 0.9 1* graph, where a speedup of ~80 was achieved on 4 cores. In this graph, the parallel algorithm was clearly able to prune away significantly more branches of the search tree, due to sharing the bounding condition between the threads, than its sequential counterpart.

In two graphs, *hamming 8-2* and *hamming 10-2*, the parallel speedups were <1 for all numbers of cores and decreased with increasing number of cores (Figure 9). There were also less extreme cases where speedup increased almost linearly up to certain number of cores but stagnated or even decreases at higher numbers of cores, *e.g.*, in *MANN a25* and *MANN a45* graphs in which speedup increased to ≈ 4 on up to four cores, then stagnated, and fell to ≈ 3 at 24 cores. This undesired effect may be related to a very high density and very large maximum cliques of these graphs (see Table ST1), causing the parallel algorithm to prune less branches than the sequential algorithm.

In the majority of graphs, however, speedup increases with the number of cores. In Figure 9, a general trend in speedup depending on the execution time of the sequential algorithm can be seen. Speedups are higher for graphs that are harder to solve, *i.e.*, in which the sequential algorithm needs more time to find the maximum clique. For example, in the *hamming 8-2* graph that is solved very quickly, the parallelization overhead consisting of sequential preprocessing, are the main causes for a slowdown. Conversely, in harder graphs, such as *p hat500-3*, speedup increases almost linearly with each additional core. In such graphs, even the 12 additional virtual cores provided by hyperthreading increase the speedup, although less than the physical cores (experiments on up to 12 cores use all physical cores while the experiments on 24 cores use 12 physical and 12 virtual cores). These results clearly indicate that MaxCliquePara algorithm is significantly faster than two comparable algorithms on most DIMACS graphs, and that on some graphs it achieves speedups of up to two orders on multiple cores.

Protein Product Graphs

Results on a single core show that MaxCliquePara algorithm outperforms the other three tested algorithms on most protein product graphs (Table 2). The difference in execution times is especially large in favor of our algorithm on *2w00B-3h1tA* and *3hrzA-2hr0A* graphs, where MaxCliquePara's is almost 80 times faster than BBMC algorithm. The Bron-Kerbosch algorithm performs better on protein product graphs than on DIMACS graphs. It is the fastest on two graphs; however, it is orders of magnitude slower than the three maximum clique algorithms on all other graphs.

[b] Super-linear speedup is the speedup greater than k, on k cores. It can be achieved if one or more caches are integrated with individual cores, increasing total cache size available to the algorithm, allowing faster execution on the account of faster memory access. Super-linear speedup can also be achieved if the parallel algorithm traverses the search space using a more efficient trajectory than its corresponding sequential algorithm.

Table 2. Execution times on protein product graphs on a single core using four exact maximum clique algorithms, with results given as a mean of 15 trials ± standard deviation

Graph name	Number of amino acids	n[a]	p[b]	Clique size	Execution time [s]			
					MaxCliqueSeq	MaxCliqueDyn	BBMC	Bron-Kerbosch
3zy0D-3zy1A	<50	61	0,98	52	**0.000205 ± 0.000064**	0.000244 ± 0.000066	0.000217 ± 0.00009	> 2h
3p0kA-3gwlB	200-300	138	0,94	89	**0.000896 ± 0.000071**	0.00171 ± 0.00009	0.000996 ± 0.00038	> 2h
2uv8I-2j6iA	>2000	200	0,86	69	**0.00327 ± 0.00019**	0.00866 ± 0.000075	0.00441 ± 0.00033	> 2h
1f82A-1zb7A	500-1000	271	0,99	247	**0.0104 ± 0.00097**	0.0401 ± 0.00017	0.0127 ± 0.00093	3330 ± 6.7
2w4jA-2a2aD	300-400	563	0,98	447	0.0693 ± 0.00073	0.38 ± 0.0012	0.0809 ± 0.0018	**0.0353 ± 0.0011**
1kzkA-3kt2A	50-100	451	0,97	346	0.0695 ± 0.0062	0.193 ± 0.0038	**0.0616 ± 0.0054**	5470 ± 18
2w00B-3h1tA	1000-1500	346	0,91	143	**0.0894 ± 0.00013**	0.465 ± 0.0085	6.38 ± 0.095	0.368 ± 0.0068
1allA-3dbjC	100-200	655	0,97	500	0.113 ± 0.0053	0.606 ± 0.0088	0.669 ± 0.0086	**0.000372 ± 0.000016**
2fdvC-1po5A	400-500	750	0,96	556	**0.259 ± 0.002**	0.938 ± 0.0017	0.378 ± 0.00095	> 2h
3hrzA-2hr0A	1500-2000	905	0,94	563	5.66 ± 0.072	**4.43 ± 0.011**	444 ± 1.8	> 2h

[a] *n* is number of vertices in a graph.

[b] *p* is graph density.

[c] Fastest execution time in each row is in bold.

The results on multiple cores show favorable speedups achieved by the MaxCliquePara algorithm (Figure 10). Speedups reached their peaks at 5 to 8 threads, but additional threads decreased speedups. For example, in graphs *2w00B-3h1tA* and *3hrzA-2hr0A*, MaxCliquePara achieved high speedups of 14 and 43 when using 6 and 7 threads, respectively. In both cases speedup decreased when additional threads were added. Speedups were higher on larger protein product graphs, suggesting that our algorithm is most suitable for comparison of larger protein structures composed of several hundreds of amino acids. These results indicate that MaxCliquePara algorithm is very suitable for use in protein structural comparisons.[2]

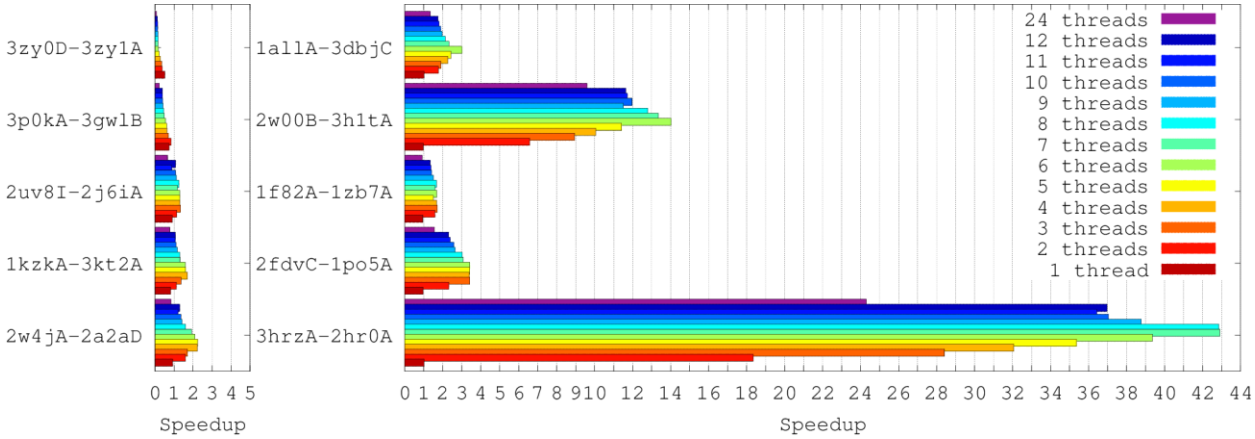

Figure 10. Speedup of the MaxCliquePara algorithm on protein product graphs on multiple cores. Graphs are sorted by the increasing execution time on a single core (left to right, top to bottom). Exact values for speedups are given in Supporting Information in Table ST3.

Conclusion

The newly developed exact parallel maximum clique algorithm MaxCliquePara is presented. It demonstrates an efficient solution of maximum clique problems and outperforms most widely used sequential algorithms on general DIMACS and protein-derived benchmark graphs on a single and on multiple cores. The performed experiments show significant speedups of the MaxCliquePara algorithm for lower numbers of parallel cores on most tested graphs. With up to the maximum 12 available physical cores, and even with the additional 12 hyperthreading-enabled cores (a total of 24 cores), the speedup scales great on larger and more computationally demanding graphs. An important advantage of MaxCliquePara is its use of the shared memory parallelism, which enables small overhead, and thus fast execution on wide range of graph sizes. Super-linear speedups are observed, consistent with expectations for an algorithm that traverses a tree of possible solutions. For graphs that are "easy" for the sequential maximum clique algorithm, *i.e.*, with the execution time in order of milliseconds, the parallelism brings no significant benefits. The speedups are, however, almost always greater than one, therefore, the MaxCliquePara algorithm can currently be taken as one of the fastest general maximum clique algorithms for almost all but the most trivial graphs.

ASSOCIATED CONTENT

Supporting Information. Freely available codes of our implementation for all maximum clique algorithms used here. Statistics of DIMACS graphs are in Table ST1. Detailed speedups for MaxCliquePara parallel maximum clique algorithm are presented in Tables ST2-ST3. This material is available free of charge via the Internet at http://pubs.acs.org.

AUTHOR INFORMATION

Corresponding Author

To whom correspondence should be addressed. E-mail: dusa@cmm.ki.si

Notes

The authors declare no competing financial interest.

Author Contributions

‡These authors contributed equally.

Funding Sources

The financial support through grant P1-0002 of the Slovenian Research Agency is acknowledged.

REFERENCES

1. Karp, R. M. *Reducibility among combinatorial problems*. Plenum Press: New York, 1972.
2. Konc, J.; Janezic, D. ProBiS algorithm for detection of structurally similar protein binding sites by local structural alignment. *Bioinformatics* **2010,** *26*, 1160-1168.
3. Eblen, J.; Phillips, C.; Rogers, G.; Langston, M. The maximum clique enumeration problem: algorithms, applications, and implementations. *BMC Bioinformatics* **2012,** *13*, S5.
4. Barker, E. J.; Buttar, D.; Cosgrove, D. A.; Gardiner, E. J.; Kitts, P.; Willett, P.; Gillet, V. J. Scaffold hopping using clique detection applied to reduced graphs. *J. Chem. Inf. Model.* **2006,** *46*, 503-511.
5. Butenko, S.; Wilhelm, W. E. Clique-detection models in computational biochemistry and genomics. *Eur. J. Oper. Res.* **2006,** *173*, 1-17.
6. Artymiuk, P. J.; Poirrette, A. R.; Grindley, H. M.; Rice, D. W.; Willett, P. A graph-theoretic approach to the identification of three-dimensional patterns of amino acid side-chains in protein structures. *J. Mol. Biol.* **1994,** *243*, 327-344.
7. Artymiuk, P. J.; Poirrette, A. R.; Rice, D. W.; Willett, P. The use of graph theoretical methods for the comparison of the structures of biological macromolecules. In *Molecular Similiarity II*, Sen, K. D., Ed. Springer: Berlin Heidelberg, 1995; pp 73-103.
8. Konc, J.; Janezic, D. ProBiS: a web server for detection of structurally similar protein binding sites. *Nucleic Acids Res.* **2010,** *38*, W436-W440.
9. Irwin, J. J.; Shoichet, B. K. ZINC-a free database of commercially available compounds for virtual screening. *J. Chem. Inf. Model.* **2005,** *45*, 177-182.
10. Raymond, J. W.; Gardiner, E. J.; Willett, P. Rascal: Calculation of graph similarity using maximum common edge subgraphs. *Comput. J.* **2002,** *45*, 631-644.
11. Raymond, J. W.; Gardiner, E. J.; Willett, P. Heuristics for similarity searching of chemical graphs using a maximum common edge subgraph algorithm. *J. Chem. Inf. Comp. Sci.* **2002,** *42*, 305-316.
12. Raymond, J. W.; Willett, P. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. *J. Comput. Aided Mol. Des.* **2002,** *16*, 521-533.
13. Prosser, P. Exact algorithms for maximum clique: A computational study. *Algorithms* **2012,** *5*, 545-587.
14. Carraghan, R.; Pardalos, P. M. An exact algorithm for the maximum clique problem. *Oper. Res. Lett.* **1990,** *9*, 375-382.
15. Tomita, E.; Sutani, Y.; Higashi, T.; Takahashi, S.; Wakatsuki, M. A simple and faster branch-and-bound algorithm for finding a maximum clique. In *WALCOM: Algorithms and computation*, Md. Saidur, R.; Satoshi, F., Eds. Springer: Berlin Heidelberg, 2010; Vol. 5942, pp 191-203.
16. Konc, J.; Janezic, D. An improved branch and bound algorithm for the maximum clique problem. *MATCH Commun. Math. Comput. Chem.* **2007,** *58*, 569-590.
17. Fahle, T. Simple and fast: Improving a branch-and-bound algorithm for maximum clique. *Lect. Notes. Comput. Sc.* **2002,** *2461*, 485-498.
18. San Segundo, P.; Rodríguez-Losada, D.; Jiménez, A. An exact bit-parallel algorithm for the maximum clique problem. *Comput. Oper. Res.* **2011,** *38*, 571-581.
19. Östergård, P. R. J. A fast algorithm for the maximum clique problem. *Discrete Appl. Math.* **2002,** *120*, 197-207.
20. Pardalos, P. M.; Rappe, J.; Resende, M. G. C. An exact parallel algorithm for the maximum clique problem. In *High Performance Algorithms and Software in Nonlinear Optimization*, Kluwer Academic Publishers: Netherlands, 1998; pp 279-300.

21. McCreesh, C.; Prosser, P. Distributing an Exact Algorithm for Maximum Clique: maximising the costup. *arXiv preprint arXiv:1209.4560* **2012**.
22. Konc, J.; Janezic, D. ProBiS-2012: web server and web services for detection of structurally similar binding sites in proteins. *Nucleic Acids Res.* **2012,** *40*, W214-W221.
23. Segundo, P.; Matia, F.; Rodriguez-Losada, D.; Hernando, M. An improved bit parallel exact maximum clique algorithm. *Optim. Lett.* **2013,** *7*, 467-479.
24. Konc, J.; Janezic, D. Protein-protein binding-sites prediction by protein surface structure conservation. *J. Chem. Inf. Model.* **2007,** *47*, 940-944.
25. Rose, P. W.; Bi, C.; Bluhm, W. F.; Christie, C. H.; Dimitropoulos, D.; Dutta, S.; Green, R. K.; Goodsell, D. S.; Prlic, A.; Quesada, M.; Quinn, G. B.; Ramos, A. G.; Westbrook, J. D.; Young, J.; Zardecki, C.; Berman, H. M.; Bourne, P. E. The RCSB Protein Data Bank: new resources for research and education. *Nucleic Acids Res.* **2013,** *41*, D475-D482.
26. Biggs, N. Some heuristics for graph coloring. In *Graph Colourings*, Longman: New York, 1990; pp 87-96.
27. Wood, D. R. An algorithm for finding a maximum clique in a graph. *Oper. Res. Lett.* **1997,** *21*, 211-217.
28. Tomita, E.; Kameda, T. An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. *J. Global Optim.* **2007,** *37*, 95-111.
29. Tsai, C. C.; Johnson, M.; Nicholson, V.; Naim, M. A topological approach to molecular similarity analysis and its application. In *Studies in Physical and Theoretical Chemistry*, Elsevier: USA, 1987; Vol. 51, pp 231-236.
30. Janezic, D.; Miličević, A.; Nikolić, S.; Trinajstić, N. *Graph-Theoretical Matrices in Chemistry*. University of Kragujevac: Kragujevac, 2007; Vol. 3.
31. Dijkstra, E. W. Solution of a problem in concurrent programming control. *Commun. ACM* **1965,** *8*, 569.
32. Bron, C.; Kerbosch, J. Algorithm 457: finding all cliques of an undirected graph. *Commun. ACM* **1973,** *16*, 575-577.