J. Parallel Distrib. Comput. 69 (2009) 417–428

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

A scalable, parallel algorithm for maximal clique enumeration☆

Matthew C. Schmidt [a,b], Nagiza F. Samatova [a,b,*], Kevin Thomas [c], Byung-Hoon Park [b]

[a] Computer Science Department, North Carolina State University, Raleigh, NC 27695, United States
[b] Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN 37831, United States
[c] Cray, Inc., United States

ARTICLE INFO

Article history:
Received 5 July 2008
Received in revised form
19 November 2008
Accepted 20 January 2009
Available online 30 January 2009

Keywords:
Maximal clique enumeration
Parallel graph algorithms
High-performance computing
Dynamic load balancing
Biological networks
Cray XT

ABSTRACT

The problem of maximal clique enumeration (MCE) is to enumerate all of the maximal cliques in a graph. Once enumerated, maximal cliques are widely used to solve problems in areas such as 3-D protein structure alignment, genome mapping, gene expression analysis, and detection of social hierarchies. Even the most efficient *serial* MCE algorithms require large amounts of time to enumerate the maximal cliques in networks arising from these problems that contain hundreds, thousands, or larger numbers of vertices. The previous attempts to provide practical solutions to the MCE problem through *parallel* implementation have had limited success, largely due to a number of challenges inherent to the nature of the MCE combinatorial search space. On the one hand, MCE algorithms often create a *backtracking* search tree that has a highly irregular and hard-or-impossible to predict structure; therefore, almost any static decomposition of the search tree by parallel processors results in highly unbalanced processor execution times. On the other hand, the data-intensive nature of the MCE problem often makes naive dynamic load distribution strategies that require extensive data movement prohibitively expensive. As a result, good scaling of the overall execution time of parallel MCE algorithms has been reported for only up to a couple hundred processors. In this paper, we propose a parallel, scalable, and memory-efficient MCE algorithm for distributed and/or shared memory high performance computing architectures, whose runtime scales linearly for thousands of processors on real-world application graphs with hundreds and thousands of nodes. Its scalability and efficiency are attributed to the proposed: (a) representation of the search tree decomposition to enable parallelization; (b) parallel depth-first backtracking search to both constrain the search space and minimize memory requirement; (c) least stringent synchronization to minimize data movement; and (d) on-demand work stealing intelligently coupled with work stack splitting to minimize computing elements' idle time. To the best of our knowledge, the proposed parallel MCE algorithm is the first to achieve a linear scaling runtime using up to 2048 processors on Cray XT machines for a number of real-world biological networks.

Published by Elsevier Inc.

1. Introduction

The maximal clique enumeration (MCE) problem is the problem of enumerating all maximal cliques in a graph. Informally, a graph or undirected graph $G = (V, E)$ is a set of vertices V and a set of edges E that connect pairs of distinct vertices. A *clique* in G is a subset of vertices in the graph G where each pair of vertices in the clique is connected by an edge in G. The size of a clique is defined as the number of vertices in the clique. A *maximal clique* in G is a clique whose vertices are not all contained in some larger clique. Additionally, a clique is defined as maximal if there is no vertex in V that is adjacent to all of the vertices in the clique. A maximal clique should not be confused with the *maximum* clique, which is the largest clique in the graph. Fig. 1 gives an example of maximal and maximum cliques in a graph with 6 vertices and 10 edges. The maximal cliques in the graph are the sets of vertices $\{A, C, E\}$, $\{B, C, E, F\}$, and $\{C, D, F\}$. The maximum clique in the example graph is the vertex set $\{B, C, E, F\}$.

The MCE problem is known to be *NP*-hard [25] that, in practice, translates to exponential (or worse) runtimes with respect to the number of vertices. The runtime of an MCE algorithm will depend on the number of maximal cliques in an input graph, since

☆ The authors are thankful to the reviewers for their insightful comments and to Cray Inc. for the access to large-scale Cray XT systems and the insights into the code optimization and benchmarks. This research has been supported by the "Exploratory Data Intensive Computing for Complex Biological Systems" project from US Department of Energy (Office of Advanced Scientific Computing Research, Office of Science). The work of MCS was also sponsored by the Dean of the Department of Computer Science at North Carolina State University. The work of NFS was also sponsored by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory. Oak Ridge National Laboratory is managed by UT-Battelle for the LLC US D.O.E. under contract no. DEAC05-00OR22725.

* Corresponding author at: Computer Science Department, North Carolina State University, Raleigh, NC 27695, United States.
E-mail address: samatovan@ornl.gov (N.F. Samatova).

0743-7315/$ – see front matter. Published by Elsevier Inc.
doi:10.1016/j.jpdc.2009.01.003

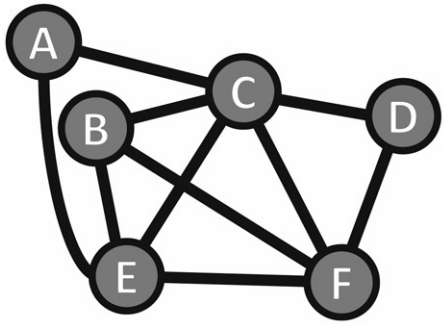


Fig. 1. An example graph.

Table 1
The worst-case and actual number of maximal cliques in 4 biological networks. The description of these networks can be found in Section 2.3.

Graph	Vertices	Edges	Cliques	
			Worst-Case	Actual
`ava80`	193,568	2,260,872	$10^{30.785}$	395,306
`fedex6`	4,088	518,395	10^{650}	400,788,124
`rmat`	10,000	1,000,000	10^{1590}	15,837,141
`yeast7`	3,472	246,342	10^{552}	2,644,041,089

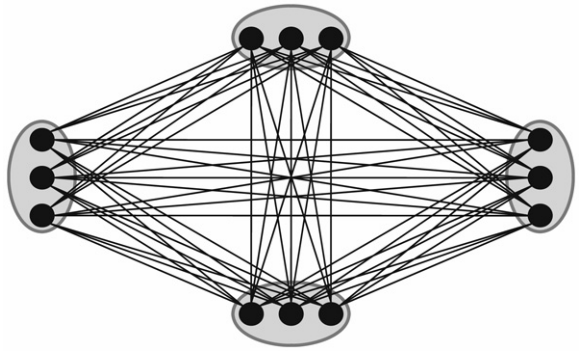


Fig. 2. A Moon–Moser graph.

the solution of the MCE problem requires the enumeration of all maximal cliques. In the worst case, the number of maximal cliques will be $3^{n/3}$, where n is the number of vertices in the graph [30]. Fig. 2 shows an example of such a graph with 12 vertices. However, graphs that model real-world data (such as the world wide web, social networks, and biological networks) are unlikely to exhibit this worst case behavior (Table 1).

There are a number of applications that require the generation of all maximal cliques in a graph. For example, the solutions of the MCE problem are used to align 3-dimensional protein structures [12], to integrate genome mapping data [18], to identify co-expressed genes [35], to identify common secondary structure elements of proteins [17], to find clusters of orthologous genes [33], to detect protein–protein interaction complexes [41], to cluster similar mass spectrometry spectra [37], and to detect social hierarchy from e-mail communications [36]. The challenge remains of how to scale an MCE algorithm to graphs with hundreds, thousands, or even millions of vertices, which are not unusual for the problems considered in these examples.

A parallel method of generating maximal cliques could allow the enumeration of maximal cliques in graphs of the desired scale, especially with the emerging trends in computer architectures with millions of processors. However, creating an efficient parallel MCE algorithm for such architectures is a non-trivial problem. Embarrassingly, parallel methods of enumerating maximal cliques are often unable to handle the data-intensive nature of enumerating maximal cliques in large graphs [40].

1.1. Major contributions

The parallel algorithm presented in this paper efficiently handles the data-intensive nature of the MCE problem and has a speedup that is linear even when thousands of processors are used to run the algorithm on real-world biological networks with thousands and hundreds of thousands of vertices, such as those derived from data on protein–protein interactions and evolutionary gene sequence conservation across multiple species data. To the best of our knowledge this is the first parallel MCE algorithm that scales to this number of processors on such large-scale real-world problem sizes.

The proposed parallel algorithm is a parallelization of the widely-used method of enumerating maximal cliques of Bron and Kerbosch (*BK*) [10]. An effective decomposition of the *BK* search tree into independent search subtrees whose leaf nodes represent maximal cliques is proposed. To allow this decomposition, candidate path data structures containing the minimal information required for a *BK* search subtree exploration are introduced. A candidate path also contains sufficient information to further constrain the search space via backtracking search.

To enable efficient and scalable parallelization, a dynamic load balancing scheme is proposed. The unbalanced depth of the search subtrees explored by individual processors will often lead to the unbalanced termination times of the processors despite an evenly balanced initial workload distribution. To alleviate this problem, a dynamic or runtime work stealing process [24] intelligently coupled with a stack splitting procedure [16] guarantees a minimal idle time of each processor. As a result, the processor utilization efficiency is maximized and linear scaling with the number of processors is achieved.

The implementation of the parallel algorithm is designed to be neutral with respect to the hardware on which it is run. The algorithm uses POSIX threads for shared memory access and MPI data transfer for distributed memory access. The algorithm can be run on distributed memory, shared memory, or "hybrid" systems. This is in line with current high-performance computing trends toward multi-core architectures. The software is available upon request.

The rest of the paper is organized as follows. In Section 2 a review of previous serial and parallel MCE algorithms is given. Section 3 introduces the *BK* algorithm and its decomposition through the use of *candidate path* structures. Section 4 describes the parallel MCE algorithm and presents the motivation and results of various design decisions. And in Section 5 there is a discussion of the benefits of the scalable, parallel MCE algorithm presented in this paper as well as some future applications of this work.

2. Background

2.1. Serial MCE algorithms

The MCE problem originally manifested itself as a set of related problems in diverse research areas. Algorithms for solving these problems can be thought of as the first MCE algorithms. The widely accepted first attempt at maximal clique enumeration is the algorithm of Harary and Ross [19] published in 1957. This algorithm was presented as a method of finding clique relationships between people using sociometric data. The MCE algorithm of Paull and Unger [34] and its improvement by Marcus [29] were constructed for the problem of minimizing rows in a flow table. The MCE algorithm presented by Bonner [8] is motivated by a desire to cluster data to assist with the classification of diseases. In addition, the algorithm of Bednarek and Taulbee [7] assists with the construction of maximal chains.

As the MCE problem began to arise in more and more research areas, a more concerted effort was made to create an efficient algorithm for enumerating all maximal cliques in a graph. The initial attempts at this followed what [32] refers to as the *point removal method*. In this approach, the cliques of a graph G are built from the cliques in the graph $G \setminus \{v\}$, in which the vertex v is removed along with its incident edges from the graph G. The early algorithm of Bonner [8] uses the point removal method. In 1970 Augustson and Minker [4] determined that an unpublished point removal algorithm by Bierstone [9] offered a significant time improvement over the algorithm of Bonner [8]. A correction to the version of the Bierstone algorithm given by Augustson and Minker in [4] was published by Mulligan and Corneil in 1972 [31]. A major problem with point removal algorithms is that all of the cliques in the graph $G \setminus \{v\}$ must be stored in memory while generating the cliques of the graph G. This can lead to enormous memory requirements.

MCE algorithms achieved a significant speedup with the introduction of *backtracking* search techniques for the enumeration of the maximal cliques. Backtracking improves the efficiency of a combinatorial optimization algorithm by limiting the size of the search space of the algorithm. It does this by establishing a set of viability criteria that are used to keep the backtracking algorithm from expanding unpromising search paths. In 1973, two backtracking algorithms were introduced to solve the MCE problem. One, an algorithm by Akkoyunlu [2], recursively divides the search space such that every maximal clique in the original search space was contained in exactly one of the new, smaller search spaces. The other, the algorithm by Bron and Kerbosch [10] (*BK*), uses a data structure to remember which search paths it has already visited so as not to revisit them. Both algorithms only explore search paths that are guaranteed to contain a maximal clique. Also, because both algorithms are implemented using a stack structure, the memory requirements of the algorithms is polynomial in the size of the graph.

Since 1973, the *BK* algorithm has proved far more pervasive than the algorithm of Akkoyunlu. This is partly due to the relative simplicity of the *BK* algorithm implementation when compared to the algorithm of Akkoyunlu. The work of Johnston [20] compared the original *BK* algorithm with a number of different variations of the *BK* algorithm. It was found that the original *BK* algorithm performed well against all of the variations. Later, the backtracking algorithms of Loukakis and Tsouros [27] and Loukakis [26] have shown to be more efficient than the *BK* for small graphs of up to 220 vertices. However, it is unclear how the performance of *BK* algorithm and these algorithms compare when they are run on large graphs.

Recent MCE work has focused on developing algorithms that use a *reverse search* strategy of enumerating maximal cliques. Avis and Fukuda describe reverse search algorithms in [5], which define a parent relationship for an enumerable object such that every object has exactly one parent except for the root. This relationship can be used to define a tree structure. Reverse search algorithms enumerate objects by traversing this tree.

One key feature of reverse search algorithms, in addition to their increased efficiency, is that it is often possible to define an upper bound on their runtime as a polynomial with respect to the number of maximal cliques in the graph. One of the first reverse search algorithms for MCE was the algorithm of Tsukiyama et al. [39]. Other reverse search algorithms include Chiba and Nishiseki [13], Johnson et al. [21], and two from Makino and Uno presented in [28]. The upper bounds of the time and space complexity of these algorithms are given in Table 2. In Table 2 n is the number of vertices in G, m is the number of edges in G, μ is the number of maximal cliques in G, $a(G)$ is the arboricity of G (definition in [13]), $M(n)$ is the time needed to multiply two $(n \times n)$ matrices, and Δ is the maximum vertex degree of G.

Table 2
The upper bound on the time and space complexity of reverse search based algorithms.

Paper	Runtime	Space
Tsukiyama et al. [39]	$O(nm\mu)$	$O(n+m)$
Chiba and Nishiseki [13]	$O(a(G)m\mu)$	$O(m)$
Johnson et al. [21]	$O(n(m+n\log\mu))$	$O(n\mu)$
Makino and Uno [28] (Matrix multiplication)	$O(M(n)\mu)$	$O(n^2)$
Makino and Uno [28] (Sparse graphs)	$O(\Delta^4\mu)$	$O(n+m)$

Table 3
The rate of clique output for the serial version of *BK* algorithm for the 4 biological networks described in Section 2.3.

Graph	Average degree	Density	Cliques/second
ava80	23.4	0.0001	122,842
fedex6	253.6	0.06	350,952
rmat	200.0	0.02	373,879
yeast7	141.9	0.04	504,215

In practice, the most efficient serial clique enumeration algorithms have been shown to be related to *BK*. In [22], Koch presented a modification to the order in which the search paths were explored by the *BK* algorithm. In experimental runs of the Koch and *BK* algorithms, the Koch algorithm occasionally ran up to eight times faster than the *BK* algorithm. However, it is worth noting that changes in the implementation of *BK* can lead to similar speedup. For instance, in [40] Zhang et al. show a 32 to 64 times improvement in the running time of *BK* due to the use of a bit-wise representation of the data. Tomita et al. [38] presented an algorithm that was a slight modification of the Koch algorithm. This algorithm was shown to run orders of magnitude faster than the algorithms of Makino and Uno for several random graphs. The algorithm by Tomita et al. output cliques at a rates between 5000 and 1,000,000 cliques per second for random graphs. As can be seen in Table 3 the serial *BK* algorithm implemented in this paper output cliques at comparable rates. The results in Table 3 were obtained by running our parallel algorithm on a single Cray XT4 processor.

2.2. Previous parallel MCE algorithms

The previously developed *pClique* [40], the first parallel MCE algorithm, extends the work of Kose et al. [23] (dubbed as KOSE). In principle, KOSE is identical in spirit to the *BK* algorithm; it branches with the alphanumerical ordering. However, whereas *BK* is a recursive algorithm with depth-first search (DFS) branching, KOSE is a serial algorithm with breadth-first search (BFS) branching (see [14] for DFS and BFS definitions). This property allows cliques of size k to be generated from cliques of size $k-1$, sharing the flavor of a famous association rule mining algorithm, *a priori* [1]. Consequently, all maximal cliques are produced in non-decreasing order. Such a property can be an invaluable asset to certain applications. Nevertheless, BFS branching strategy inevitably makes KOSE memory-intensive. Although *pClique* improves KOSE performance by utilizing bit-vector manipulation of common neighbors, huge memory requirements remain unchanged. This affects both the size of the graphs that can be handled by the *pClique* algorithm and the speedup achieved by using more processors. The authors of [40] describe an experimental run of the *pClique* algorithm on a graph consisting of 2895 vertices and 10,914 edges using an SGI Altix 3700 machine. Using 256 processors, the *pClique* algorithm achieved at most a speedup factor of 91. This motivates the development of a parallel DFS-based *BK* algorithm.

Another parallel MCE algorithm is the *Peamc* algorithm presented in [15] by Du et al.. The *Peamc* algorithm does not suffer

from the same memory liabilities as *pClique* because it is based on a serial DFS-based MCE procedure, also presented in [15]. This serial MCE procedure generates a disjoint set of maximal cliques for every vertex in a given graph. *Peamc* uses a simple parallelization scheme that simply assigns each parallel process a disjoint set of vertices and allows each process to enumerate every maximal clique in the set of maximal cliques assigned to it by the serial algorithm. Without an additional load balancing step after the initial work distribution, certain processes would be allowed to terminate early. This can be seen by experimental data provided in [15]. Using a cluster of 30 3.2 GHz processors with 2 GB of main memory on each node, the *Peamc* algorithm enumerates the maximal cliques in a set of 10 graphs each with around 50,000 vertices, 1,000,000 edges and 150,000 maximal cliques. The *Peamc* algorithm only achieves a speedup factor of 23 with 30 processors in the best case. This motivates our use of a dynamic load balancing scheme in the parallel DFS-based *BK* algorithm.

2.3. Description of experimental networks

Results of the execution of the parallel algorithm presented here were gathered from the execution of the parallel algorithm on a number of different real-world graphs. They are referred to by their shorthand names `ava80`, `rmat`, `fedex6`, and `yeast7`. What follows is a description of how each of these different graphs was obtained.

The `ava80` graph contains 193,568 vertices and 2,260,872 edges. The vertices represent genes from 80 bacterial genomes, where 40 genomes are of aerobic phenotypic type and the rest of the genomes are of anaerobic phenotypic type. Two genes are connected if their sequences are sufficiently similar. Specifically, for every pair of genes, their sequence similarity is measured by computing their BLAST E-value [3]. The edges in the `ava80` graph connect pairs of genes in two different species whose BLAST E-value is less than a threshold of 10^{-5}.

The `rmat` graph has 100,000 vertices and 2,000,000 edges. The graph is synthetically generated using *GTgraph* [6] and follows the Recursive Matrix Graph model (R-MAT) [11] so that it has a scale-free nature.

The `fedex6` graph contains 4088 vertices and 518,395 edges. The vertices represent genes in the *Shewanella oneidensis* MR-1 genome and the vertices are connected by an edge if their gene expression levels are correlated. Specifically, the gene expression level of a single gene is measured over 12 different microarray experiments. For each pair of genes, the Pearson correlation coefficient is calculated using these gene expression levels. The edges in the `fedex6` graph connect pairs of genes whose correlation coefficient is above 0.6.

The `yeast7` graph has 3472 vertices and 246,342 edges. The vertices represent genes in the yeast *Saccharomyces cerevisiae* genome, and the vertices are connected if their gene expression levels are correlated. Specifically, the gene expression level of a single gene is measured over 127 different microarray experiments under different conditions. For each pair of genes, the Pearson correlation coefficient is calculated using these gene expression levels. The edges in the `yeast7` graph connect pairs of genes whose correlation coefficient is above 0.7.

3. Decomposition of serial MCE algorithm

In order to create a scalable, memory-efficient, parallel MCE algorithm, an efficient serial MCE algorithm was selected that was suitable for parallelization. The algorithm was then decomposed into multiple tasks that could be easily distributed. The well-known algorithm of Bron and Kerbosch (*BK*) [10] was chosen as the efficient serial MCE instance. The selection of *BK* led to the development of the *candidate path* structure (Section 3.2), which allowed the task of generating a search node in the *BK* search tree to be independent of the task of generating its children in the search tree. Particular care was taken during the parallel implementation of the *BK* algorithm to improve the efficiency of determining connectivity between two vertices in a graph, because this task is performed repeatedly by *BK*. The following subsections describe the motivation of each of these decisions.

3.1. The BK method of maximal clique enumeration

The parallel MCE algorithm generates maximal cliques in a method similar to *BK* [10]. A pseudocode description of *BK* can be found in Fig. 3(a) and (b). A quick overview of *BK* is given to highlight aspects of the method that are important in the construction of the parallel algorithm.

BK enumerates maximal cliques by using backtracking search to visit all of the vertices in a maximal clique. The search paths in *BK* are expanded into a search tree structure. An example of the search tree structure generated when running *BK* on the graph in Fig. 3(c) is shown in Fig. 3(d). A search path can be extended in *BK* by visiting an unexplored vertex that is connected to every vertex already explored by the search path. Thus, the explored vertices form a clique in the graph. If a search path cannot be expanded, then the representative clique is a maximal clique.

A traversal of the search tree generated by *BK* is performed by the `CliqueEnumerate` function shown in Fig. 3(b), which requires three lists of vertices each time it visits a vertex of the graph.

(i) (`compsub` list) A list of vertices in the current search path;
(ii) (`cand` list) A list of candidate vertices that are not in (i) but are connected to every vertex in (i);
(iii) (`not` list) A list of vertices that are connected to every vertex in (i) but would form a redundant, or previously enumerated, search path if combined with the vertices in (i).

When the `CliqueEnumerate` function visits a vertex, it adds the vertex to the `compsub` list and constructs new `cand` and `not` lists. The new `cand` list is constructed by finding all of the vertices in the `cand` list that are connected to every vertex in the `compsub` list including the newly added vertex. The new `not` list is constructed by finding all of the vertices that are in the `not` list and connected to every vertex in the `compsub` list. In Fig. 3(d) the state of the `compsub`, `cand`, and `not` lists are given in the bottom three rows, respectively, of the rectangles comprising the nodes of the search tree in the figure. The first row of the rectangle represents the vertex of the graph being visited by that instance of the `CliqueEnumerate` function.

The `CliqueEnumerate` function expands a search path by visiting the vertices in the `cand` list. The first vertex visited by the `CliqueEnumerate` function is the vertex in the `cand` list that is connected to the most other vertices in the `cand` list. After this first vertex, only vertices in the `cand` list that are not connected to this first vertex are visited. This is done to ensure that the fewest number of vertices are visited for each call of the `CliqueEnumerate` function. Once a vertex in the `cand` list has been visited, it is moved to the `not` list. If there are no vertices in the `cand` list or the `not` list, then the vertices in the `compsub` structure represent a maximal clique and the vertices are output as such (the black rectangles in Fig. 3(e)). Once the `CliqueEnumerate` function has visited all of its children vertices, it backtracks to the previous vertex in the search path. As further explained in Section 3.2, the backtracking step can be implemented through the use of a stack structure (Fig. 3(e)).

Using the *BK* method of enumerating maximal cliques in the parallel algorithm has a number of advantages. As mentioned in Section 2.1 *BK* is a fast MCE algorithm that is widely used

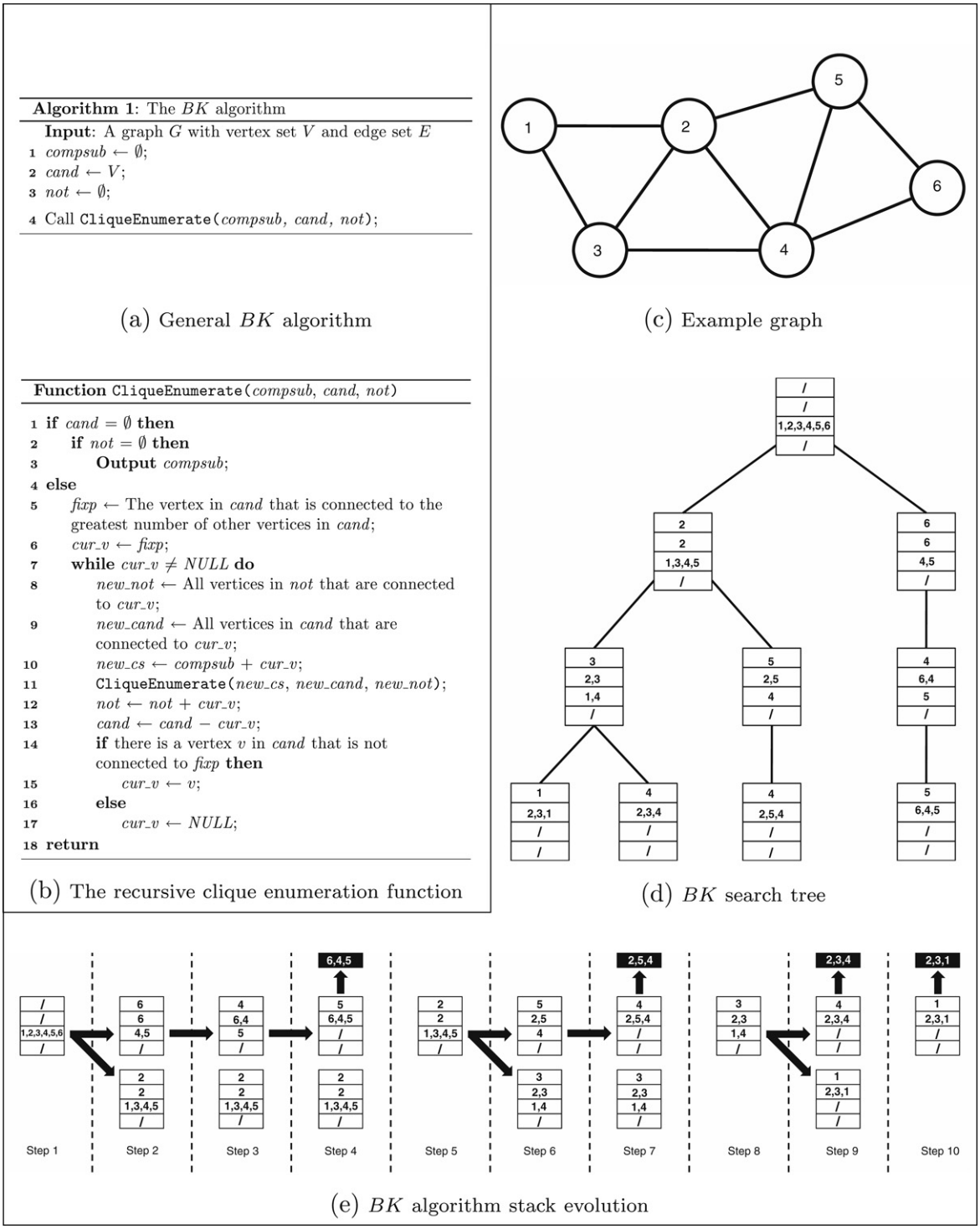


Fig. 3. The *BK* algorithm.

because it is efficient while being relatively simple to implement. The search process of the few MCE algorithms known to be faster than BK, such as the algorithms of Koch [22] and Tomita et al. [38], use a backtracking search process similar to *BK*. Thus, parallel algorithms based on these MCE methods could apply parallelization methods presented in this paper.

3.2. Designing candidate path data structure

Parallelization, the process of converting a single serial execution into multiple serial threads, requires that the overall task be decomposed into multiple tasks, which are ideally independent. The overall task of a backtracking algorithm like *BK* is to traverse the search tree. This single task can be decomposed into obviously independent tasks of traversing unexplored search subtrees. This makes a parallel implementation of *BK* seem straightforward. However, different search subtrees may differ greatly in size. In order to ensure that the exploration of the search tree is divided evenly among compute nodes, the task of traversing an unexplored search subtree may need to be dynamically reassigned among compute nodes. In a parallel implementation, the reassignment of this task requires an explicit representation of the unexplored

search subtree that can be copied among compute nodes with minimal data transfer (Fig. 14).

In the *BK* algorithm, a search subtree can be completely and exactly traversed if the `compsub`, `cand`, and `not` lists of the root node of the search subtree are known, as well as the graph data. All of the maximal cliques found in a *BK* search subtree will contain every vertex in the subtree root's `compsub` list as well as a subset of vertices from the subtree root's `cand` list. No other vertex can be part of a maximal clique. Knowing the vertices in the subtree root's `compsub` and `cand` lists will allow the algorithm to generate any possible maximal clique found in the *BK* search subtree. As can be seen in Fig. 3, the *BK* search also limits expansion of its subtrees by not visiting any vertex in the `not` list and bounding the subtree when a vertex in the `not` list is connected to every vertex remaining in the `cand` list. Therefore, knowing the subtree root's `not` list will ensure that no extra vertices are visited.

To enable parallelization by explicitly representing an unexplored search subtree, we introduce the *candidate path* data structure. For any given search node in the *BK* search tree, a candidate path data structure stores the vertex being visited by the search node, the level the search node exists in the *BK* search tree (with the root of the *BK* search tree being at level-0), and an explicit representation of both the `cand` and `not` lists of the search node. The `compsub` list for a given subtree root is not stored in the candidate path data structure because it is stored as a global array in each computing element that is updated throughout the traversal of the search tree. For example, if the root of the *BK* search tree is at level-0, a forward traversal step in the search algorithm that goes from level-0 to level-1 in the *BK* search tree adds a vertex at index-0 in the global `compsub` array. A backtracking step in the search algorithm that goes from level-2 to level-1 in the *BK* search tree removes the vertex stored at index-1 from the global `compsub` array (Fig. 3(e)).

When the work of exploring a given search subtree is transferred between computing elements, the `compsub`, `cand`, and `not` lists of the subtree root search node are transferred. The `compsub` list of the search node is reconstructed from the global `compsub` array, the vertex visited by the search node, and the level of the search tree stored in the candidate path structure. Thus, the data in the candidate path structure can be considered an explicit representation of the task of exploring a given search subtree.

3.3. Improving serial algorithm efficiency

In the *BK* algorithm, a connected predicate is used repeatedly to determine if the current vertex shares an edge with a test vertex. Since this operation occurs many times, it should have an efficient implementation. Three possible methods of implementing this operation were considered.

1. A linear search of a linked list of adjacent vertices
2. A lookup using an adjacency bit matrix
3. A lookup using a hash table of edges.

The simplest implementation is a linear search of the list of vertices adjacent to the current vertex, since this list is part of the basic graph description. While this uses no additional memory, it is also the slowest method. It is especially bad for a common case, where the test vertex is not in the list. The fastest method is to construct an adjacency matrix from the graph description. This is done by creating a $|V| \times |V|$ matrix and entering a true value in row i and column j if there is an edge between vertices i and j in the graph description and a false value for every other entry in the matrix. Determining if vertices i and j are connected is done by examining the (i, j) entry in the bit matrix. A limitation of this technique is that a matrix for a graph with a large number of vertices may not fit in the memory. A fast technique that often requires only a limited amount of extra memory is a hash table.

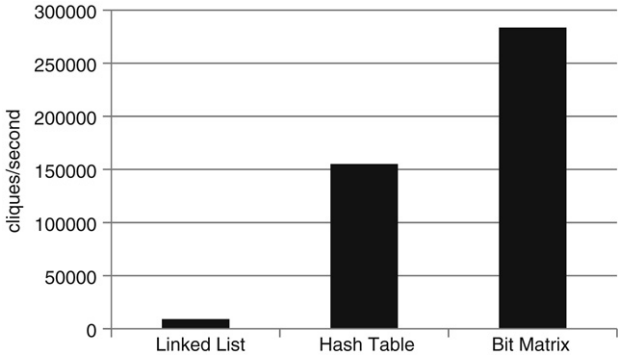


Fig. 4. A comparison of the number of cliques output per second when the parallel MCE algorithm was run using different methods of determining vertex connectivity.

A compromise between a linked list and a bit matrix approach is a lookup procedure in a hash table of adjacent vertices. The size of the table is a small constant factor larger than the list representation, and the average lookup requires fewer memory accesses. A simple hash function (the test vertex number modulo the table size) is used to compute its index into the current vertex's hash table row. A table collision occurs when two vertices have the same hash index. In the worst case, the hash table lookup time becomes linear in the vertex degree if all vertices have the same hash index. To avoid this, a size expansion factor is used to allocate a table several times larger than the vertex degree when the table is created. A sparsely populated hash table has much less chance of encountering the worst case. In the average case, the lookup time is constant with respect to the vertex degree. The memory required for the hash table grows with the number of edges in the graph. The hash table approach is applicable when the average vertex degree is much smaller than the number of vertices; otherwise, the adjacency matrix is more compact.

The parallel algorithm implements all three adjacency test methods. A comparison of the speed of the three methods of determining vertex connectivity during a run of the algorithm on the `yeast7` graph using a single processor on a Cray XT4 machine is given in Fig. 4. It uses the adjacency matrix if enough memory is available, defaulting to the hash table for large graphs. If insufficient memory for the hash table is available, the adjacent vertex list search is used. An alternative to the linear search of the adjacency list is to sort the list and use binary search when the list size (or sub-list size) is large. For the graph sizes and compute node sizes considered, the lookup techniques are the best choice. If the graph sizes of interest grow substantially, or the compute nodes used are significantly smaller, binary search may be the preferred algorithm.

4. Parallelization of MCE algorithm

Though the introduction of candidate path structures allows the task of generating maximal cliques according to the *BK* method to be decomposed into independent subtasks, it does not make the task embarrassingly parallel. The work required to complete the subtask of exploring subtrees of the *BK* search tree is not uniform. In addition, it is impossible to estimate the amount of work involved in a subtask *a priori*. This means that there will be a need to dynamically reallocate subtasks among computing elements in order to ensure that all available computing elements are utilized during the execution of the parallel algorithm. This is achieved by a dynamic load balancing (DLB) procedure.

If p is the number of processors, then the runtime of the parallel algorithm can roughly be described by the following equation:

$$T(p) = T_{\text{init}} + \max_{1 \le i \le p}(T_{\text{enum}}(i) + T_{\text{idle}}(i)). \quad (1)$$

The time T_{init} represents the time spent to serially initialize the parallel algorithm. This could include the time needed to read in the input graph and the time needed to distribute the initial work. The time $T_{\text{enum}}(i)$ represents the time spent by a computing element enumerating maximal cliques in the given graph. The time $T_{\text{idle}}(i)$ represents any time that a computing element spends not enumerating maximal cliques. This could include time spent on load balancing and termination procedures.

To reduce the size of the term $\max_{1 \le i \le p}(T_{\text{enum}}(i) + T_{\text{idle}}(i))$, the amount of time $T_{\text{enum}}(i) + T_{\text{idle}}(i)$ should be equal for each computing element, meaning that each should terminate at the same time. If this is the case, then the total time of the parallel algorithm can be written as:

$$T(p) = T_{\text{init}} + \frac{T_{\text{enum}} + T_{\text{idle}}}{p} \quad (2)$$

where

$$T_{\text{enum}} = \sum_{1 \le i \le p} T_{\text{enum}}(i) \quad (3)$$

$$T_{\text{idle}} = \sum_{1 \le i \le p} T_{\text{idle}}(i). \quad (4)$$

Ideally, the parallel algorithm will achieve a linear speedup, where doubling the number of computing elements used by the parallel algorithm reduces the runtime by half. The following equation gives the speedup of the parallel algorithm using p computing elements.

$$\text{Speedup}(p) = \frac{T_{\text{serial}}}{T(p)}. \quad (5)$$

The term T_{serial} refers to the runtime of the serial version of the algorithm. In this paper, we are more concerned with the relative speedup of the parallel algorithm, especially when the serial algorithm takes an impractical amount of time. The formula for relative speedup is given as follows.

$$\text{Speedup}_{\text{rel}}(p) = \frac{T(p)}{T(2p)}. \quad (6)$$

In the total runtime of the parallel algorithm, the times T_{init} and T_{idle} are usually dependent on p, the number of computing elements used by the algorithm, and the time T_{enum} is independent of p. The parallel algorithm must do three things to achieve a linear speedup: (1) it must minimize $T_{\text{idle}}(i)$; (2) it must minimize T_{init}; and (3) it must ensure that all computing elements terminate at nearly the same time.

4.1. Overview of parallel algorithm

A brief overview of the parallel algorithm is given here, and the pseudocode is given in Figs. 5 and 6. The algorithm initially reads in and distributes the input graph information such that every computing element has access to a copy (Section 4.2.1 and lines 1–7 of Fig. 5) because the subtask of generating a subtree in the *BK* search tree requires that a computing element be able to check the adjacency of vertices in the input graph. Once the graph data has been distributed, all candidate path structures that represent cliques of size-1 are generated and distributed to the computing elements (Section 4.2.2 and lines 1–9 of Fig. 6). Once a computing element has these size-1 candidate path structures, it can begin to independently generate maximal cliques according to the *BK* method of generating maximal cliques (Section 3 and lines 11–15 of Fig. 6).

The computing elements continue to generate maximal cliques independently until one of the elements finishes exploring all of the subtrees of the *BK* search for which it had been assigned.

```
1  if process_id = 0 then
2      Read in graph data;
3      Construct vertex adjacency list;
4      Broadcast vertex adjacency list;
5  else
6      Wait for vertex adjacency list broadcast;
7      Construct linked list, adjacency matrix, or
       hash table;
8      for i = 1; i < num_local_threads; i++ do
9          Spawn thread T_i;
10         Have T_i run CliqueEnumerate();
11     Have T_0 run CliqueEnumerate();
12     Join all local threads;
13     foreach other process P_j do
14         Send P_j an idle message;
15     Wait for idle message from every other
       process;
```

Fig. 5. Pseudocode for the parallel MCE algorithm.

```
1  foreach vertex v_i assigned to the thread do
2      cp ← New candidate path structure for v_i;
3      foreach vertex v_j in the graph do
4          if connected(v_i, v_j) then
5              if i < j then
6                  Vertex v_j is in cp's cand list;
7              else
8                  Vertex v_j is in cp's not list;
9      Push cp onto cp_stack;
10 while cp_stack is not empty do
11     cur ← Pop a candidate path from cp_stack;
12     if cur's cand and not lists are empty then
13         Output thread's compsub;
14     else
15         Generate all of cur's children;
16     if there are MPI requests then
17         tmp ← StealWorkFromLocalThread();
18         Send MPI response with tmp;
19 tmp ← StealWorkFromLocalThread();
20 if tmp ≠ NULL then
21     Push tmp onto cp_stack;
22     goto line 10;
23 if local_thread_id ≠ 0 then
24     Make thread idle;
25 else
26     work_received ← false;
27     foreach local thread T_i do
28         foreach other process P_j do
29             Send MPI request to P_j;
30             if P_j's MPI response has work then
31                 Give T_i work;
32                 work_received ← true;
33                 break;
34 if local_thread_id ≠ 0 then
35     Restart thread;
36 if work_received = true then
37     goto line 10;
```

Fig. 6. Pseudocode for the parallel algorithm's `CliqueEnumerate` function.

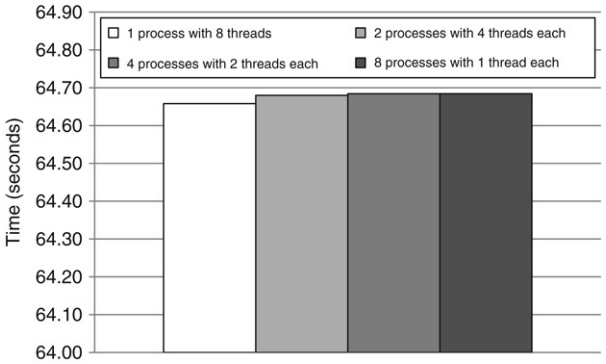


Fig. 7. A comparison of the amount of time required by the MCE algorithm when run using 8 different computing elements is divided 4 different ways.

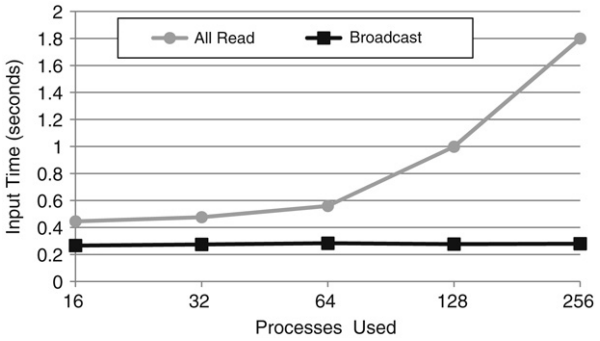


Fig. 8. A comparison of the amount of time required by a parallel I/O approach to distributing the graph data and an MPI broadcast message.

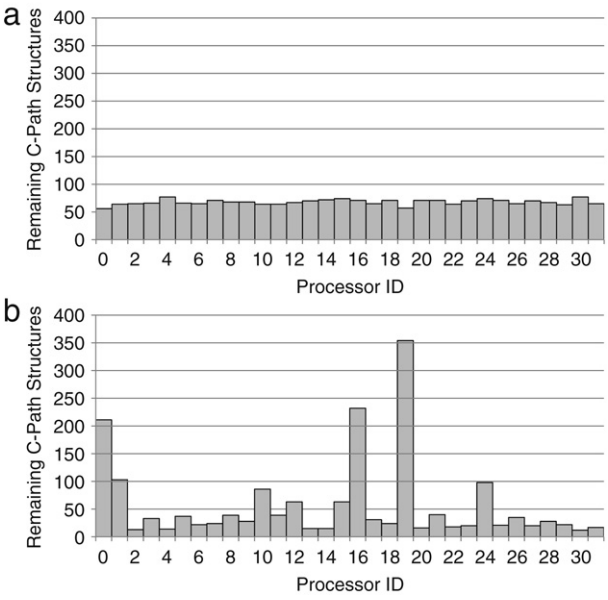


Fig. 9. (a) Represents the size of the initial loads. (b) Represents the size of the loads after 5 s of execution of the parallel algorithm.

At this point the idle computing element requests for more subtrees to explore from another randomly chosen computing element (Section 4.2.3 and lines 16–37 of Fig. 6). This is referred to as *random polling* and is one of the most efficient methods of requesting work when the underlying architecture of the computing system is not known [24]. If the responding computing element has work to send, it responds with candidate path structures that are most likely to represent large subtrees. This is motivated by the *stack splitting* procedure presented by Finkel and Manber in [16]. If the responding computing element has no work then the requesting computing element selects a different computing element and tries again. If all of the computing elements are idle, then the program terminates (Section 4.2.4 and lines 13–15 of Fig. 5).

Parallel execution of the program is achieved by generating multiple *processes* that are each capable of spawning multiple *threads*. Inter-process communication is performed using MPI communication, and the threaded behavior of the application is enabled using POSIX threads (Pthreads). Each process is assumed to have its own memory, that its associated threads share. This "hybrid" parallelism is motivated by the fact that many modern high-performance machines consist of clusters of symmetric multiprocessing (SMP) units. By combining both shared-memory and distributed-memory parallelism techniques, the best performance can be achieved. Flexibility is preserved by allowing the application to use multiple processes with multiple threads, a single process with multiple threads, multiple processes with a single thread, or a single process with a single thread, depending on the target computer system. A comparison of the usage of threads and processors during a run of the algorithm on the `yeast7` graph on a Cray XT4 machine is given in Fig. 7.

4.2. Algorithm design details

4.2.1. Reading input graph

Every computing element requires access to a copy of the graph data. Multiple threads in a single process can share access to the graph data in memory, but each process requires its own copy of the graph data. A parallel I/O approach where all processes read the graph description file is limited by file system scalability. With large numbers of processes, it is more efficient to have a single process read the input file and broadcast the graph data to all of the other processes using a call to `MPI_Bcast` (Fig. 8).

The graph data is represented as an adjacency list as it is read in and broadcast to the other processes. The adjacency list for each vertex is managed as an array rather than a linked list. This makes the graph data more compact, results in better cache behavior when the adjacency list is accessed, and also eliminates address pointers, which are not portable between processes and would need to be regenerated locally after the broadcast. Auxiliary representations of the graph data, such as the adjacency hash table or adjacency bit matrix discussed in Section 3.3, are generated by each process. It is cheaper to generate this data locally than to transfer it through the network.

4.2.2. Initial work distribution

As Section 4.2.3 will explain in greater detail, it is impossible to determine a well-balanced distribution of subtasks in the parallel algorithm *a priori*. As seen in Fig. 9, taken from a run of the parallel algorithm on the `fedex6` graph on a Cray XT3 machine using 32 processes, even an initially balanced distribution of work can become uneven. Therefore, a dynamic load balancing procedure will always be required to ensure computing elements do not become idle prematurely. Therefore, the communication overhead of the initial work distribution procedure will be a greater factor in the scalability of the algorithm than its ability to produce an initially balanced workload.

To distribute the initial work, computing nodes must be assigned the subtasks of exploring the subtrees in the *BK* search tree whose roots represent size-1 cliques. Each vertex in the graph represents a size-1 clique. Therefore, the initial work of the parallel algorithm can be distributed by dividing the vertices of the graph amongst the computing nodes. Given this partitioning scheme, this step is done independently by each thread and no communication

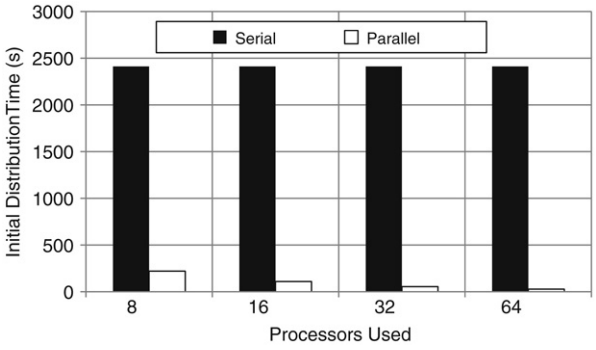


Fig. 10. A graph of the time to generate the initial work using serial and parallel methods.

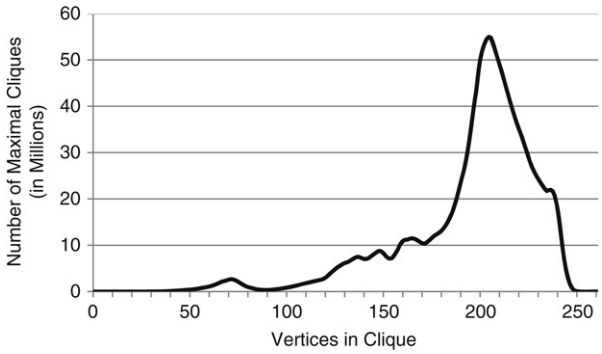


Fig. 11. The distribution of maximal cliques in the `yeast7` graph with respect to clique size.

between processes is required. Each thread selects root vertices based on its global thread number, which is computed using the local thread number and the MPI process rank. Allowing each thread to compute its own initial work instead of having a single process distribute the initial work to all of the other processes further parallelizes the overall algorithm. Fig. 10 shows the benefit of such an approach. In a run of the parallel algorithm on the `ava80` graph on a Cray XT3 machine, the initial work distribution times of the serial algorithm are hugely disproportionate to the time required by the algorithm to enumerate the maximal cliques, which was less than a second for all four runs.

4.2.3. Dynamic load balancing

The nature of the *BK* search tree makes it impossible to determine *a priori* a well balanced distribution of subtasks. The subtask defined by the decomposition of the *BK* algorithm is the task of generating all of the search nodes in a subtree rooted at a given search node. The amount of work involved in this subtask can vary because it depends on the size and number of maximal cliques in the subtree. As can be seen in Fig. 11 the size of maximal cliques in a given graph can vary widely. Furthermore, because of the way *BK* generates maximal cliques, certain subtrees will generate many more maximal cliques than others. However, neither the size nor the number of maximal cliques in a given subtree of the *BK* search tree can be predicted without generating the entire subtree. Thus, certain "unlucky" computing elements may be assigned subtrees with larger and more maximal cliques than others. Without allowing these overloaded computing nodes to transfer work to underloaded ones, the execution time of the different computing elements may differ greatly without load balancing. An example of this is given in Fig. 12, which was generated during runs of the parallel algorithm on the `fedex6` graph on a Cray XT3 machine using 16 processes with 1 thread each.

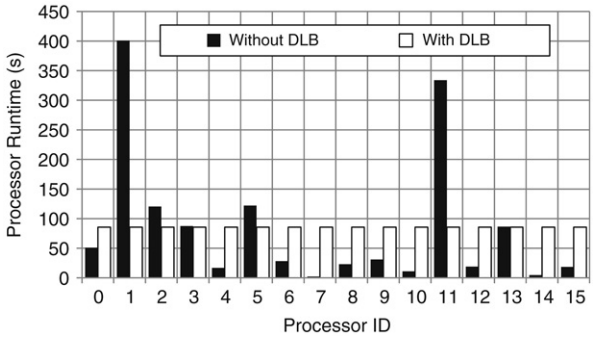


Fig. 12. The different execution times of various processors when the dynamic load balancing procedure was used (white bars) and not used (black bars).

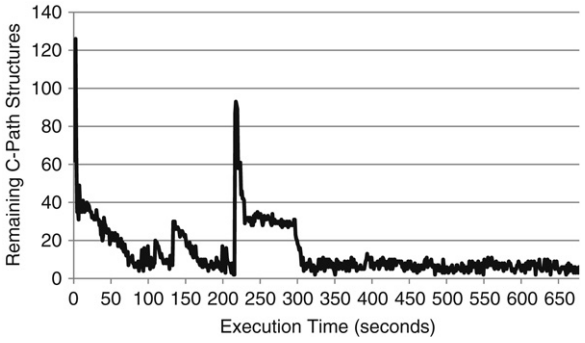


Fig. 13. The number of candidate path (C-Path) structures remaining in a given processor's stack during the execution of the parallel algorithm.

The dynamic load balancing scheme aims to maximize the overall processor utilization efficiency or equivalently to minimize each individual processor's idle (non-computing) time. The parallel algorithm allows work stealing to keep computing elements from becoming idle for significant amounts of time. The amount of work that a computing element has is measured by the number of candidate path structures left in its stack. A receiver-initiated dynamic load balancing scheme was chosen because the number of candidate paths remaining in a stack tends to decrease gradually and increase rapidly over the course of the algorithm's execution. This can be seen in Fig. 13, which was generated during a run of the parallel algorithm on the `fedex6` graph on a Cray XT3 machine using 32 different processes with 1 thread each. It is hard to predict which computing elements will become idle until they actually become idle, because their workload can increase quickly. Therefore, it is more efficient to allow a computing element to become almost idle (stack size less than a threshold) and then send requests for more work.

Two levels of load balancing are used, corresponding to the levels of parallelization. Within a process, threads can steal from one another by simply removing an item from the pool. Locks are used to prevent the owner thread, or another stealing thread, from accessing the pool during this operation. This local load balancing is used until all threads of the process have no work.

When all local threads have no work, a more complex load balancing scheme is needed for a thread to steal work from a thread in another process. A load balance request message is sent to a random target process, which responds with any items that it wishes to share. The term stealing still applies, since the responding target process must steal the work from local threads in order to send it to the requesting process. In order to make the remote stealing efficient, more than one item may be stolen per request. The request message specifies the number of idle threads in the requesting process, and up to that many items may be returned.

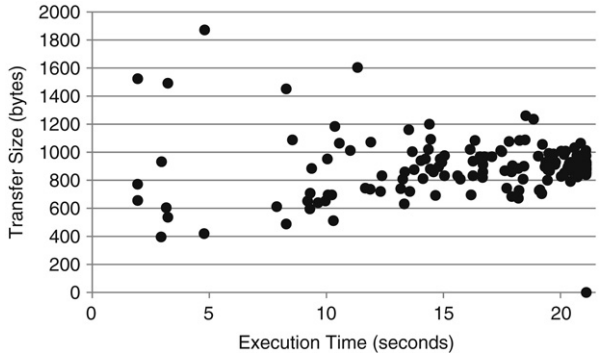


Fig. 14. A plot of the amount of data transferred to a given process and the time during the execution the transfer occurred.

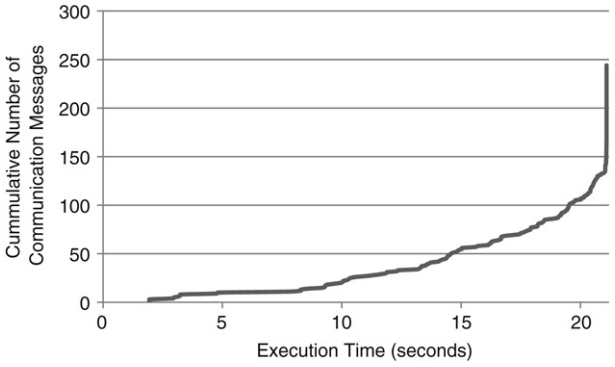


Fig. 15. A representative example of the cumulative number of interprocess communication messages sent by a given processor during the execution of the parallel algorithm.

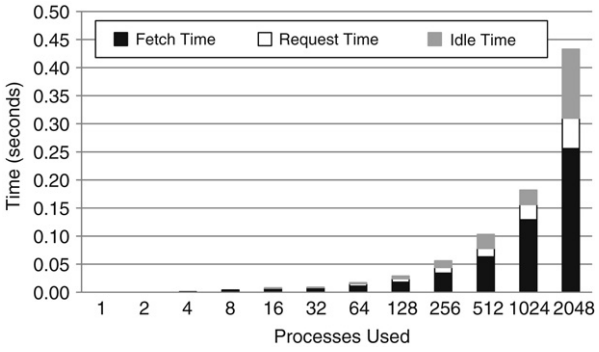


Fig. 16. MPI communication overhead times.

In both cases, local and remote, if an attempt to steal work fails because the work pools are empty, the search continues to the next target, until all targets (threads or processes) have been polled.

The candidate paths transferred between requesting and receiving processes always come from the bottom of the stack of candidate paths, while the candidate paths being examined always come from the top of the stack. This *stack splitting* technique is used because the candidate path structures at the bottom of the stack represent unexplored search subtree rooted at higher levels of the search tree. These search subtrees are more likely to generate work than those at the lower levels of the search tree, as seen in Fig. 14 a graph generated during an execution of the parallel algorithm run with 64 different processes with 1 thread each on the `fedex6` graph on a Cray XT3 machine.

4.2.4. Termination condition

Since the enumeration proceeds without global synchronization, a technique to complete processing in an orderly way is needed. The first issue is that an individual process must poll all other processes to determine that no process has work to share, i.e. no further load balancing can occur. Secondly, each process must handle load balancing requests until it can determine that all other processes have reached the same point. This implies that a process cannot terminate (i.e., call `MPI_Finalize`) even when no further work is available.

Shutdown, therefore, takes place in phases. First a process polls all other processes for work to share. If all requests fail, the process is ready to enter an idle state. In the idle state, it continues to respond to load balancing requests, but first it sends an idle status notification to every other process. When idle notifications are received from all other processes, then the entire application has reached a quiescent point and all processes can terminate.

While this procedure generates a flurry of communication during the termination process, the effect on overall application performance has not been significant. This can be seen in Fig. 15, an example generated from a run of the parallel algorithm with 64 different processes with 1 thread each on the `fedex6` graph on a Cray XT3 machine. If this simple broadcast scheme of communication becomes a performance issue, either at an extreme scale of compute nodes or on a particular computer system, then hierarchical communication may be necessary by partitioning the processes into smaller groups.

4.3. Scalability results

The goal of the parallelization methods presented here is to create an algorithm whose runtime scales linearly with the number of computing elements used to run the parallel algorithm. As mentioned in the beginning of Section 4, whether or not the parallel algorithm scales linearly depends on limiting the size of T_{init} and T_{idle}. The time T_{init} depends only on the time needed to read in the input graph, broadcast the data using the network, and set up the additional data structures due to the design decisions discussed in Sections 4.2.1 and 4.2.2. The time T_{idle} is determined by the time spent by the various processors either requesting work, transferring work, or simply being idle. The size of T_{idle} can be seen in Fig. 16, which was generated during a run of the parallel algorithm on the `yeast7` graph using between 1 and 2048 processes on a Cray XT4. Although these times are only a small percentage of the overall run time for the benchmark runs, the trend shows that the overhead may become significant at extreme scale (10s of thousands of processes).

The relative small size of T_{idle} means that the parallel algorithm will likely scale linearly as long as T_{init} remains small. Figs. 17 and 18 show that the speedup of the algorithm is linear and thus the initialization and idle times are small relative to the total execution time. Fig. 17 was generated by runs of the parallel algorithm on the `fedex6`, `ava80`, and `rmat` graphs using between 16 and 1024 processes on a Cray XT4 machine. The number of processors was increased by a factor of 2 until the runtime of the parallel algorithm was less than a second. Fig. 18 was generated by runs of the parallel algorithm on the `yeast7` graph with between 1 and 2048 processes on a Cray XT4.

5. Conclusions and future work

The parallel algorithm presented in this paper is able to handle the data-intensive nature of the MCE problem for large graphs, while also having a runtime that scales linearly with the number of processors used. The algorithm utilizes the *BK* method of enumerating maximal cliques. The candidate path structure is introduced to enable the decomposition of the *BK* method of

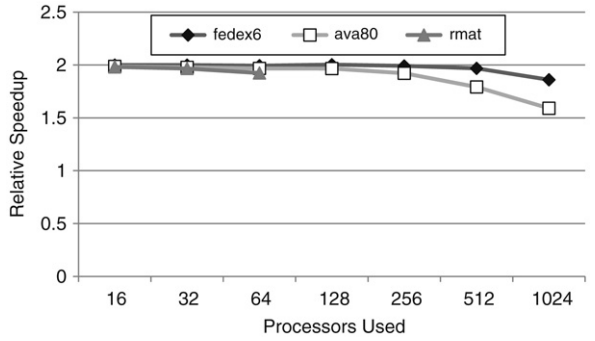


Fig. 17. The relative speedup of the parallel algorithm.

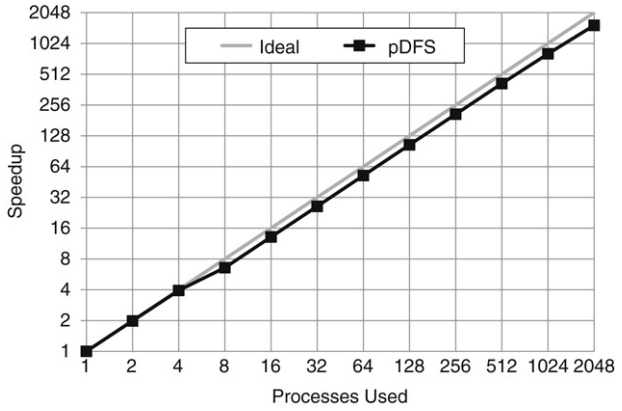


Fig. 18. The speedup of the parallel algorithm. The viewgraph is presented in $\log_2$ scale.

enumerating maximal cliques. This decomposition allows for work to be redistributed according to a proposed dynamic load balancing procedure that keeps all processes busy during the execution of the parallel algorithm without introducing significant communication overhead. An initial work distribution was generated in parallel, as opposed to being controlled in serial by a master process. This eliminated a bottleneck in the execution of the parallel algorithm and improved the efficiency of generating the initial work of using the *BK* method. This non-trivial parallelization framework allows the runtime of the parallel algorithm to scale linearly with the number of processors used by the algorithm.

Current research has shown that the parallel algorithm can be used to enumerate the maximal cliques in large-scale biological networks. However, the algorithm remains to be tested on other large-scale real-world networks. Networks derived from non-biological data could exhibit different properties than biological networks. These properties may affect the practical scalability of the parallel algorithm.

The imprecision of real-world (e.g., biological) data from which the networks are derived means that obtaining useful information about the networks may require the solutions to problems similar to the MCE problem instead of the MCE problem itself. For instance, protein–protein interaction data is often represented as a network with edge probabilities. If it is not known which interaction probabilities should be considered significant, then it may be desirable to enumerate all of the maximal cliques in a set of graphs whose edges have an interaction probability greater than a sliding threshold. Algorithms that do not require the re-enumeration of all of the maximal cliques for a different threshold are needed. Likewise, algorithms that enumerate "maximal weighted cliques" may be of more practical value.

References

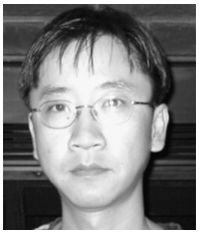
[1] R. Agrawal, R. Srikant, Fast algorithms for mining association rules, in: Proc. 20th Int. Conf. Very Large Data Bases, VLDB, 1994, pp. 487–499.
[2] E.A. Akkoyunlu, The enumeration of maximal cliques of large graphs, SIAM Journal on Computing 2 (1) (1973) 1–6.
[3] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, D.J. Lipman, Basic local alignment search tool, Journal of Molecular Biology 215 (3) (1990) 403–410.
[4] J.G. Augustson, J. Minker, An analysis of some graph theoretical cluster techniques, Journal of the ACM 17 (4) (1970) 571–588.
[5] D. Avis, K. Fukuda, Reverse search for enumeration, Discrete Applied Mathematics 65 (1–3) (1996) 21–46.
[6] D.A. Bader, K. Madduri, GTgraph: A synthetic graph generator suite, 2006, http://www.cc.gatech.edu/~kamesh/GTgraph/.
[7] A.R. Bednarek, O.E. Taulbee, On maximal chains, Revue Roumaine de Mathematiques pures et Appliquees 11 (1) (1966) 23–25.
[8] R.E. Bonner, On some clustering techniques, IBM Journal of Research and Development 8 (1) (1964) 22–32.
[9] E. Brierstone, Cliques and generalized cliques in a finite linear graph, Unpublished Report.
[10] C. Bron, J. Kerbosch, Algorithm 457: Finding all cliques of an undirected graph, Communications of the ACM 16 (9) (1973) 575–577.
[11] D. Chakrabarti, Y. Zhan, C. Faloutsos, R-mat: A recursive model for graph mining, in: Proc. of the 4th SIAM Int. Conf. on Data Mining, SDM, 2004, pp. 442–446.
[12] Y. Chen, G.M. Crippen, A novel approach to structural alignment using realistic structural and environmental information, Protein Science 14 (12) (2005) 2935–2946.
[13] N. Chiba, T. Nishizeki, Arboricity and subgraph listing algorithms, SIAM Journal on Computing 14 (1) (1985) 210–223.
[14] T. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 2nd ed., McGraw-Hill, 2001.
[15] N. Du, W. Bin, X. Liutong, W. Bai, P. Xin, A parallel algorithm for enumerating all maximal cliques in complex network, in: Proc. of ICDM Workshops, 2006, pp. 320–324.
[16] R. Finkel, U. Manber, Dib-a distributed implementation of backtracking, ACM Transactions on Programming Languages Systems 9 (2) (1987) 235–256.
[17] H.M. Grindley, P.J. Artymiuk, D.W. Rice, P. Willett, Identification of tertiary structure resemblance in proteins using a maximal common subgraph isomorphism algorithm, Journal of Molecular Biology 229 (3) (1993) 707–721.
[18] E. Harley, A. Bonner, N. Goodman, Uniform integration of genome mapping data using intersection graphs, Bioinformatics 17 (6) (2001) 487–494.
[19] F. Harary, I.C. Ross, A procedure for clique detection using the group matrix, Sociometry 20 (3) (1957) 205–215.
[20] H.C. Johnston, Cliques of a graph: Variations on the Bron-Kerbosch algorithm, International Journal of Parallel Programming 5 (3) (1976) 209–238.
[21] D.S. Johnson, M. Yannakakis, C.H. Papadimitriou, On generating all maximal independent sets, Information Processing Letters 27 (3) (1988) 119–123.
[22] I. Koch, Fundamental study: Enumerating all connected maximal common subgraphs in two graphs, Theoretical Computer Science 250 (1–2) (2001) 1–30.
[23] F. Kose, W. Weckwerth, T. Linke, O. Fiehn, Visualizing plant metabolomic correlation networks using cliquemetabolite matrices, Bioinformatics 17 (12) (2001) 1198–1208.
[24] V. Kumar, A.Y. Grama, N.R. Vempaty, Scalable load balancing techniques for parallel computers, Journal of Parallel and Distributed Computing 22 (1) (1994) 60–79.
[25] E.L. Lawler, J.K. Lenstra, A.H.G.R. Kan, Generating all maximal independent sets: NP-hardness and polynomial-time algorithms, SIAM Journal on Computing 9 (3) (1980) 558–565.
[26] E. Loukakis, A new backtracking algorithm for generating the family of maximal independent sets of a graph, Computers & Mathematics with Applications 9 (4) (1983) 583–589.
[27] E. Loukakis, C. Tsouros, A depth first search algorithm to generate the family of maximal independent sets of a graph lexicographically, Computing 27 (4) (1981) 349–366.
[28] K. Makino, T. Uno, New algorithms for enumerating all maximal cliques, in: Proc. of the 9th Scandinavian Workshop on Algorithm Theory, SWAT, 2004, pp. 260–272.
[29] M.P. Marcus, Derivation of maximal compatibles using boolean algebra, IBM Journal of Research and Development 8 (5) (1964) 537–538.
[30] J.W. Moon, L. Moser, On cliques in graphs, Israel Journal of Mathematics 3 (1) (1965) 23–28.
[31] G.D. Mulligan, D.G. Corneil, Corrections to Bierstone's algorithm for generating cliques, Journal of the ACM 19 (2) (1972) 244–247.
[32] P.M. Pardalos, J. Xue, The maximum clique problem, Journal of Global Optimization 4 (3) (1994) 301–328.
[33] B.-H. Park, N.F. Samatova, T. Karpinets, A. Jallouk, S. Molony, S. Horton, S. Arcangeli, Data-driven data-intensive computing for modelling and analysis of biological networks: Application to bioethanol production, J. Phys.: Conf. Ser. 78 (2007) 012061.
[34] M.C. Paull, S. Unger, Minimizing the number of states in incompletely specified sequential switching functions, IRE Transactions on Electronic Computers 8 (1959) 356–367.

[35] O. Rokhlenko, Y. Wexler, Z. Yakhini, Similarities and differences of gene expression in yeast stress conditions, Bioinformatics 23 (2) (2007) e184–e190.
[36] R. Rowe, G. Creamer, S. Hershkop, S.J. Stolfo, Automated social hierarchy detection through email network analysis, in: Proc. of the 9th WebKDD and 1st SNA-KDD 2007 Workshop on Web Mining and Social Network Analysis, 2007, pp. 109–117.
[37] D.L. Tabb, M.R. Thompson, G. Khalsa-Moyers, N.C. VerBerkmoes, W.H. Mcdonald, MS2Grouper: group assessment and synthetic replacement of duplicate proteomic tandem mass spectra, Journal of the American Society for Mass Spectrometry 16 (8) (2005) 1250–1261.
[38] E. Tomita, A. Tanaka, H. Takahashi, The worst-case time complexity for generating all maximal cliques and computational experiments, Theoretical Computer Science 363 (1) (2006) 28–42.
[39] S. Tsukiyama, M. Ide, H. Ariyoshi, I. Shirakawa, A new algorithm for generating all the maximal independent sets, SIAM Journal on Computing 6 (3) (1977) 505–517.
[40] Y. Zhang, F. Abu-Khzam, N. Baldwin, E. Chesler, M. Langston, N.F. Samatova, Genome-scale computational approaches to memory-intensive applications in systems biology, in: Proc. of the 2005 ACM/IEEE Conference on Supercomputing, 2005, p. 12.
[41] B. Zhang, B.-H. Park, T. Karpinets, N.F. Samatova, From pull-down data to protein interaction networks and complexes with biological relevance, Bioinformatics 24 (7) (2008) 979–986.

Matthew C. Schmidt is a doctoral student in the Department of Computer Science at North Carolina State University. He received a B.S. in Computer Science and Mathematics from Purdue University in 2004. His research interests include graph theory and algorithms, computational biology, and high-performance computing.

Nagiza F. Samatova is an Associate Professor at North Carolina State University and a Senior Research Scientist at Oak Ridge National Laboratory. She holds a Ph.D. in applied mathematics from the Computing Center of Russian Academy of Sciences (CCAS). Her research interests are in graph theory and algorithms, high performance computing, systems biology and bioinformatics, and scalable data analytics and data management.

Kevin Thomas is a member of the benchmarking staff at Cray Inc. His work focuses on high performance computer architectures, parallel algorithms, and performance analysis. His other interests include computational electromagnetics, graphical interfaces, robotics, and control systems. He received a BS in Computer Science and a B.S. in Electrical Engineering from the University of Minnesota in 1984.



Byung-Hoon Park received the M.S. and Ph.D. degree in computer science, both from Washington State University in 1996 and 2001, respectively. He is currently a research staff at Computer Science Research Group of Computer Science and Mathematics Division, Oak Ridge National Laboratory. His research area includes high-performance computing, computational biology, data mining, and computational statistics.