Efficient Implementation of the BSP/CGM Parallel Vertex Cover FPT Algorithm[⋆]

E. J. Hanashiro[1] and H. Mongelli[1] and S. W. Song[2]

[1] Universidade Federal de Mato Grosso do Sul, Campo Grande MS, Brazil,
erik@dct.ufms.br,
http://www.dct.ufms.br/∼erik,
mongelli@dct.ufms.br,
http://www.dct.ufms.br/∼mongelli,
[2] Universidade de São Paulo, São Paulo SP, Brazil,
song@ime.usp.br,
http://www.ime.usp.br/∼song

Abstract. In many applications NP-complete problems need to be solved exactly. One promising method to treat some intractable problems is by considering the so-called *Parameterized Complexity* that divides the problem input into a main part and a parameter. The main part of the input contributes polynomially on the total complexity of the problem, while the parameter is responsible for the combinatorial explosion. We consider the parallel FPT algorithm of Cheetham *et al.* to solve the k-Vertex Cover problem, using the CGM model. Our contribution is to present a refined and improved implementation. In our parallel experiments, we obtained better results and obtained smaller cover sizes for some input data. The key idea for these results was the choice of good data structures and use of the backtracking technique. We used 5 graphs that represent conflict graphs of amino acids, the same graphs used also by Cheetham *et al.* in their experiments. For two of these graphs, the times we obtained were approximately 115 times better, for one of them 16 times better, and, for the remaining graphs, the obtained times were slightly better. We must also emphasize that we used a computational environment that is inferior than that used in the experiments of Cheetham *et al.*. Furthermore, for three graphs, we obtained smaller sizes for the cover.

1 Introduction

In many applications, we need to solve NP-complete problems exactly. This means we need a new approach in addition to solutions such as approximating algorithms, randomization or heuristics.

One promising method to treat some intractable problems is by considering the so-called *Parameterized Complexity* [1]. The input problem is divided into two parts: the main part containing the data set and a parameter. For example,

[⋆] Partially supported by FAPESP Proc. No. 1997/10982-0, CNPq Proc. No. 30.5218/03-4, 30.0482/02-7, 55.2028/02-9 and DS-CAPES

in the parameterized version of the Vertex Cover problem for a graph $G = (V, E)$, also known as the k-Vertex Cover, we want to determine if there is a subset in V of size smaller than k, whose edges are incident with the vertices of this subset. In this problem, the input is a graph G (the main part) and a non-negative integer k (the parameter). For simplicity, a problem whose input can be divided like this is said to be *parameterized*.

A parameterized problem is said to be *fixed-parameter tractable*, or *FPT* for short, if there is an algorithm that solves the problem in $O(f(k)n^{\alpha})$ time, where α is a constant and f is an arbitrary function [1]. If we exchange the multiplicative connective between these two contributions by an additive connective $(f(k) + n^{\alpha})$, the definition of FPT problems remains unchanged. The main part of the input contributes polynomially on the total complexity of the problem, while the parameter is responsible for the combinatorial explosion. This approach is feasible if the constant α is small and the parameter k is within a tight interval. The k-Vertex Cover problem is one of the first problems proved to be FPT and is the focus of this work. One of the well-known FPT algorithms for this problem is the algorithm of Balasubramanian *et al.* [2], of time complexity $O(kn + 1.324718^{k}k^{2})$, where n is the size of the graph and k is the maximum size of the cover. This problem is very important from the practical point of view. For example, in Bioinformatics we can use it in the analysis of multiple sequences alignment.

Two techniques are usually applied in the FPT algorithms design: the reduction to problem kernel and the bounded search tree. These techniques can be combined to solve the problem.

FPT algorithms have been implemented and they constitute a promising approach to solve problems to get the exact solution. Nevertheless, the exponential complexity on the parameter can still result in a prohibitive cost. In this article, we show how we can solve larger instances of the k-Vertex Cover, using the CGM parallel model.

A CGM (Coarse-Grained Multicomputer) [3] consists of p processors connected by some interconnection network. Each processor has local memory of size $O(N/p)$, where N is the problem size. A CGM algorithm alternates between computation and communication rounds. In a communication round each processor can send and receive a total of $O(N/p)$ data.

The CGM algorithm presented in this paper has been designed by Cheetham *et al.* [4] and requires $O(\log p)$ communication rounds. It has two phases: in the first phase a reduction to problem kernel is applied; the second phase consists of building a bounded search tree that is distributed among the processors.

Cheetham *et al.* implemented the algorithm and the results are presented in [4]. Our contribution is to present a refined and improved implementation. In our parallel experiments, we obtained better results and obtained better cover sizes for some input data. The key idea for these results was the choice of good data structures and use of the backtracking technique. We used 5 graphs that represent conflict graphs of amino acids, and these same graphs were used also by Cheetham *et al.* [4] in their experiments. For two of these graphs, the times we obtained were approximately 115 times better, for one of them 16 times better, and, for the remaining graphs, the obtained times were slightly better. We must also emphasize that we used a computational environment that is inferior than that used in the experiments of Cheetham *et al.* [4]. Furthermore, for three graphs, we obtained smaller sizes for the cover.

In the next section we introduce some important concepts. In Section 3 we present the main FPT sequential algorithms for the problem and the CGM version. In Section 4 we present the data structures and discuss the implementation and in Section 5 we show the experimental results. In Section 6 we present some conclusions.

2 Parameterized Complexity and k-Vertex Cover Problem

We present some fundamental concepts for sequential and CGM versions of the FPT algorithm for the k-Vertex Cover problem.

Parameterized complexity [1, 5–8] is another way of dealing with the intractability of some problems. This method has been successfully used to solve problems of instance sizes that otherwise cannot be solved by other means [7].

The input of the problem is divided into two parts: the main part and the parameter. There exist some computational problems that can be naturally specified in this manner [5].

In classical computational complexity, the entire input of the problems is considered to be responsible for the combinatorial explosion of the intractable problem. In parameterized complexity, we try to understand how the different parts of the input contribute in the total complexity of the problem, and we wish to identify those input parts that cause the apparently inevitable combinatorial explosion. The main input part contributes in a polynomial way in the total complexity of the problem, while the parameter part probably contributes exponentially in the total complexity. Thus, in cases where we manage to do this, NP-complete problems can be solved by algorithms of exponential time with respect to the parameter and polynomial time with respect to the main input part. Even then we need to confine the parameter to a small, but useful, interval. In many applications, the parameter can be considered "very small" when compared to the main input part.

A *parameterizable problem* is a set $L \subseteq \Sigma^* \times \Sigma^*$, where Σ is a fixed alphabet. If the pair $(x, y) \in L$, we call x the main input part (or instance) and y the parameter.

According to Downey and Fellows [1], a *parameterizable problem* $L \subseteq \Sigma^* \times \mathbb{N}^*$ is fixed parameter tractable if there exists an algorithm that, given an input $(x, y) \in L$, solves it in $O(f(k)n^\alpha)$ time, where n is the size of the main input part x, $|x| = n$, k is the size of parameter y, $|y| = k$, α is a constant independent of k, and f is an arbitrary function.

The arbitrary function $f(k)$ of the definition is the contribution of the parameter y to the total complexity of the problem. Probably this contribution is exponential. However, the main input part contributes polynomially to the total complexity of the problem. The basic assumption is that $k \ll n$ [8]. The polynomial contribution is acceptable if the constant α is small. However, the definition of fixed parameter tractable problem remains unchanged if we exchange the multiplicative connective between the two contributions, $f(k)n^{\alpha}$, by an additive connective $f(k) + n^{\alpha}$ [1].

The fixed parameter tractable problems form a class of problems called *FPT* (*Fixed-Parameter Tractability*). There are NP-complete problems that has been proven not to be in FPT class.

An important issue to compare the performance of FPT algorithms is the maximum size for the parameter k, without affecting the desired efficiency of the algorithm. This value is called *klam* and is defined as the largest value of k such that $f(k) \leq U$, where U is some absolute limit on the number of computational steps. Downey and Fellows [1] suggest $U = 10^{20}$. A challenge in the fixed parameter tractable problems is the design of FPT algorithms with increasingly larger values of *klam*.

Two elementary methods are used to design algorithms for fixed parameter tractable problems: reduction to problem kernel and bounded search tree. The application of these methods, in this order, as an algorithm of two phases, is the basis of several FPT algorithms. In spite of being simple algorithmic strategies, these techniques do not come into mind immediately, since they involve exponential costs relative to the parameter [6].

– **Reduction to problem kernel**: The goal is to reduce, in polynomial time, an instance I of the parameterizable problem into another equivalent instance I', whose size is limited by a function of the parameter k. If a solution of I' is found, probably after an exhaustive analysis of the generated instance, this solution can be transformed into a solution of I. The use of this technique always results in an additive connective between the contributions n^{α} and $f(k)$ on the total complexity.
– **Bounded search tree**: This technique attempts to solve the problem through an exhaustive tree search, whose size is to be bounded by a function of the parameter k. Therefore, we use the instance generated by the reduction to problem kernel method in the search tree, which must be traversed until we find a node with the solution of the instance. In the worst case, we have to traverse all the tree. However, it is important to emphasize that the tree size depends only on the parameter, limiting the search space by a function of k.

In the parameterized version of the Vertex Cover problem, also known as k-Vertex Cover problem, we must have a graph $G = (V, E)$ (the instance) and a non-negative integer k (the parameter). We want to answer the following question: "Is there a set $V' \subseteq V$ of vertices, whose maximum size is k, so that for every edge $(u, v) \in E$, $u \in V'$ or $v \in V'$?". Many other graph problems can be parameterized similarly.

The set V' is not unique. An application of the vertex cover problem is the analysis of multiple sequences alignment [4]. A solution to resolve the conflicts among sequences is to exclude some of them from the sample. A conflict exists when two sequences have a score below a certain threshold. We can construct a graph, called the conflict graph, where each sequence is a vertex and an edge links two conflict sequences. Our goal is to remove the least number of sequences so that the conflict will be deleted. We thus want to find a minimum vertex cover for the conflict graph.

A trivial exact algorithm for this problem is to use brute force. In this case all the possible subsets whose size is smaller or equal to k are verified to be a cover [1], where k is the maximum size desired for the cover and n is the number of vertices in the graph ($k \leq n$). The number of subsets with k elements is $C_{n,k}$, so the algorithm to find all these subsets has time complexity of $O(n^k)$. The costly brute force approach is usually not feasible in practice.

3 FPT Algorithms for the *k*-Vertex Cover Problem

In this section we present FPT algorithms that solve the vertex cover problem and are used in our implementation. Initially we show the algorithm of Buss [9], responsible for the phase of reduction to problem kernel. Then we show two algorithms of Balasubramanian *et al.* [2] that present two forms to construct the bounded search tree. Finally we present the CGM algorithm of Cheetham *et al.* [4]. In all these algorithms, the input is formed by a graph G and the size of the vertex cover desired (parameter k).

3.1 Algorithm of Buss

The algorithm of Buss [9] is based on the idea that all the vertices of degree greater than k belong to any vertex cover for graph G of size smaller or equal to k. Therefore, such vertices must be added to the partial cover and removed from the graph. If there are more than k vertices in this situation, there is no vertex cover of size smaller or equal to k for the graph G.

The edges incident with the vertices of degree greater than k can also be removed since they are joined to at least one vertex of the cover, and the isolated vertices are removed once there are no vertices to cover. The graph produced is denominated G'.

From now on, our goal is to find a vertex cover of size smaller or equal to k' for the graph G', where k' is the difference between k and the number of elements of the partial vertex cover. This is only possible if there do no exist more than kk' edges in G'. This is because k' vertices can cover at most kk' edges in the graph, since the vertices of G' have degree bounded by k. Furthermore, if we do not have more than kk' edges in G', nor isolated vertices, we can conclude that there are at most $2kk'$ vertices in G'. As k' is at most k, the size of the graph G' is $O(k^2)$.

Given the adjacency list of the graph, the steps described until here spend $O(kn)$ time and form the basis for the reduction to problem kernel phase. Observe that graph G is reduced, in polynomial time, to an equivalent graph G', whose size is bounded by a function of the parameter k. The kernellization phase as described is used in the algorithms presented in the next subsection.

To determine finally if there exists or not a vertex cover for G' of size smaller or equal to k', the algorithm of Buss [9] executes a brute force algorithm. If a vertex cover for G' of size smaller or equal to k' exists, these vertices and the vertices of degree greater than k form a vertex cover for G of size smaller or equal to k. The algorithm of Buss [9] spends a total time of $O(kn + (2k^2)^k k^2)$.

3.2 Algorithms of Balasubramanian *et al.*

The algorithms of Balasubramanian *et al.* [2] execute initially the phase of reduction to problem kernel based on the algorithm of Buss [9]. In the second phase, a bounded search tree is generated. The two options to generate the bounded search tree are shown in Balasubramanian *et al.* [2] and described below as Algorithm B1 and Algorithm B2. In both cases, we search the tree nodes exhaustively for a solution of the vertex cover problem, by depth first tree traversal. The difference between the two algorithms is the form we choose the vertices to be added to the partial cover and, consequently, the format of such a tree.

Each node of the search tree stores a partial vertex cover and a reduced instance of the graph. This partial cover is composed of the vertices that belong to the cover. The reduced instance is formed by the graph resulting from the removal of the vertices of G that are in the partial cover, as well as the edges incident with them and any isolated vertex. We call this graph G'' and an integer k'' that is the maximum desired size for the vertex cover of G''. The root of the search tree, for example, represents the situation after the method of reduction to problem kernel. In other words, in the partial cover we have the vertices of degree greater than k and the instance $\langle G', k' \rangle$.

The edges of the search tree represents the several possibilities of adding vertices to the existing partial cover. Notice that the son of a tree node has more elements in the partial vertex cover and a graph with less nodes and edges than its parent, since every time a vertex is added to the partial cover, we remove it from the graph, together with the incident edges and any isolated vertices. We actually do not generate all the nodes before the depth first tree traversal. We only generate a node of the bounded search tree when this node is visited.

The search tree has the following property: for each existing vertex cover for graph G of size smaller or equal to k, there exists a corresponding tree node with a resulting empty graph and a vertex cover (not necessarily the same) of size smaller or equal to k. However, if there is no vertex cover of size smaller or equal to k for graph G, then no tree node possesses a resulting empty graph. Actually the growth of the search tree is interrupted when the node has a partial vertex cover of size smaller or equal to k or a resulting empty graph (case in which we find a valid vertex cover for graph G). Notice that this bounds the size of the tree in terms of the parameter k. Therefore, in the worst case, we have to traverse all the search tree to determine if there exists or not a vertex cover of size smaller or equal to k for graph G.

Given the adjacency list of the graph, we spend $O(m)$ time in each node, where m is the number of vertices of the current graph. Therefore, if $C(k)$ is the number of nodes of the search tree, then the time spent to traverse all the tree is $O(mC(k))$. Recall that the root node of the search tree, whose size is bounded by $O(k^2)$, stores the resulting graph of the phase of reduction to problem kernel.

Algorithm B1 In this algorithm, the choice of the vertices of G'' to be added to the partial cover in any tree node is done according to a path generated from any vertex v of G'' that passes through at least three edges.

If this path has size one or two, then we add the neighbor of the node of degree one to the partial cover, remove their incident edges and any isolated vertices. This new graph instance with the new partial cover is kept in the same node of the bounded search tree and the Algorithm B1 is applied again in this node.

If this path is a simple path of size three, passing by vertices v, v_1, v_2 and v_3, any vertex cover must contain $\{v, v_2\}$ or $\{v_1, v_2\}$ or $\{v_1, v_3\}$. If the path is a simple cycle of size three, passing by vertices v, v_1, v_2 and v, any vertex cover must contain $\{v, v_1\}$ or $\{v_1, v_2\}$ or $\{v, v_2\}$. In both cases, the tree node is ramified into three three sons to add one of the three pairs of suggested vertices. We can then go to the next node of the tree, recalling the depth first traversal.

Notice that this algorithm generates a tertiary search tree and that at each tree level the partial cover increases by at least two vertices. The Algorithm B1 spends $O(kn + (\sqrt{3})^k k^2)$ time to solve the k-Vertex Cover problem.

Algorithm B2 In this algorithm, the choice of vertices of G'' to be added to the partial cover in any node of the tree is done according to five cases by considering the degree of the vertices of the resulting graph. We deal first with the vertices of degree 1 (Case 1), then with vertices of degree 2 (Case 2), then with vertices of degree 5 or more (Case 3), then with vertices of degree 3 (Case 4) and, finally, with vertices of degree 4 (Case 5).

We use the following notation. $N(v)$ represents the set of vertices that are neighbors of vertices v and $N(S)$ represents the set $\bigcup_{v \in S} N(v)$.

In Case 1, if there exists a vertex v of degree 1 in the graph, then we create a new son to add $N(v)$ to the partial cover.

In Case 2, if there exists a vertex v of degree 2 in the graph, then we can have three subcases, to be tested in the following order. Let x and y be the neighbors of v. In Subcase 1, if there exists an edge between x and y, then we create a new son to add $N(v)$ to the partial cover. In Subcase 2, if x and y have at least two neighbors different from v, then we ramify the node of the tree into two sons to add $N(v)$ and $N(\{x, y\})$ to the partial cover. In Subcase 3, if x and y share an only neighbor a different from v, then we create a new son to add $\{v, a\}$.

In Case 3, if there exists a vertex of degree 5 or more in the graph, then we ramify the node of the tree into two sons to add v and $N(v)$ to the partial cover.

If none of the three previous cases occurs, then we have a 3 or 4-regular graph. In case 4, if there exists a vertex v of degree 3, then we can have four subcases, to be treated in the following order. Let x, y and z be the neighbors of v. In Subcase 1, if there exists an edge between two neighbors of v, say x and y, the we ramify the node of the tree into two sons to add $N(v)$ and $N(z)$ to the partial cover. In Subcase 2, if a pair of neighbors of v, say x and y, share another common neighbor a (but different from v), then we ramify the node of the tree into two sons to add $N(v)$ and $\{v, a\}$ to the partial cover. In Subcase 3, if a neighbor of v, say x, has at least three neighbors different from v, then we ramify the node of the tree into three sons to add $N(v)$, $N(x)$ and $x \bigcup N(\{y, z\})$ to the partial cover. In Subcase 4, the neighbors of v have exactly two private neighbors, not considering vertex v proper. Let x be a neighbor of v and let a and b be the neighbors of x, then we ramify the node of the tree into three sons to add $N(v)$, $\{v, a, b\}$ and $N(\{y, z, a, b\})$ to the partial cover.

In Case 5, we have a 4-regular graph and we can have three subcases, to be tested in the following order. Let v be a vertex of the graph and x, y, z and w its neighbor vertices. In Subcase 1, if there exists an edge between two neighbors of v, say x and y, then we ramify the node of the tree into three sons to add $N(v)$, $N(z)$ and $z \bigcup N(w)$ to the partial cover. In Subcase 2, if three neighbors of v, say x, y and z, share common neighbor a, then we ramify the node of the tree into two sons to add $N(v)$ and (v, a) to the partial cover. In Subcase 3, if each of the neighbors of v has three neighbors different from v, then we ramify the node of the tree into four sons to add $N(v)$, $N(y)$, $y \bigcup N(w)$ and $\{y, w\} \bigcup N(\{x, z\})$ to the partial cover.

Contrary to Algorithm B1, a node in the search tree can be ramified into two, three or four sons, and the partial cover can increase up to 8 vertices, depending on the selected case. Algorithm B2 spends $O(kn + 1.324718^k k^2)$ time to solve the k-Vertex Cover problem.

3.3 Algorithm of Cheetham *et al.*

The CGM algorithm proposed by Cheetham *et al.* [4] to solve the k-Vertex Cover problem parallelizes both phases of an FPT algorithm, reduction to problem kernel and bounded search tree. Previous works designed for the PRAM model parallelize only the method of reduction to problem kernel [4]. However, as the implementations of FPT algorithms usually spends minutes in the reduction to problem kernel and hours, or maybe even days in the bounded search tree, the parallelization of the bounded search tree designed in the CGM algorithm is an important contribution.

The CGM algorithm of Cheetham *et al.* [4] solves even larger instances of the k-Vertex Cover problem than those solved by sequential FPT algorithms. The implementation of this algorithm can solve instances with $k \geq 400$ in less than 75 minutes of processing time. It is important to emphasize that the k-Vertex Cover is considered well solved for instances of $k \leq 200$ (sequential FPT algorithms) [7]. Not only there is a considerable increase in the parameter k, it is important to recall that the time of a FPT algorithm grows exponentially in relation to k.

The phase of reduction to problem kernel is parallelized through a parallel integer sorting. The p processors that participate in the parallel sort are identified as P_i, $0 \leq i \leq p-1$. To identify vertices of the graph with degree larger than k, the edges are sorted by the label of the vertex they are incident with through deterministic sample sort [10], that require $O(1)$ parallel integer sorts, i.e. in constant time. The partial vertex cover (vertices with degree larger than k) and the instance $\langle G', k' \rangle$ is sent to all the processors.

The basic idea of the parallelization of the phase of bounded search tree is to generate a complete tertiary tree T with $O(\log_3 p)$ tree levels and p leaf nodes ($\gamma_0 ... \gamma_{p-1}$). Each one of these p leaf nodes is then assigned to one of the p processors, that search locally for a solution in the subtree generated from the leaf node γ_i, as shown in Fig. 1. A detailed description of this phase is presented in the following.

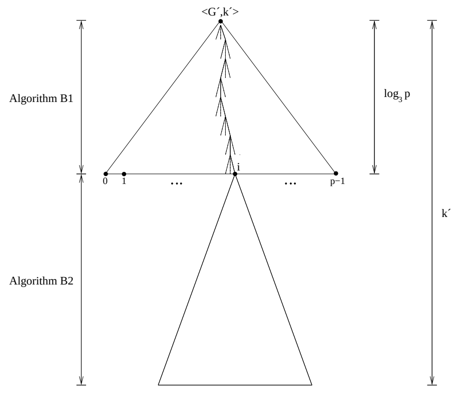

Fig. 1. A processor P_i computes the unique path in T from the root to leaf γ_i, using the Algorithm B1. Then, P_i computes the entire subtree below γ_i, using the Algorithm B2.

- Consider the tertiary search tree T. Each processor P_i, $0 \leq i < p$, starts this phase with the instance obtained at the previous phase ($\langle G', k' \rangle$), and uses Algorithm B1 to compute the unique path in T from the tree root to the leaf node γ_i. Let $\langle G_i'', k_i'' \rangle$, be the instance computed at the leaf node γ_i.
- Each processor P_i, $0 \leq i < p$, searches locally for a solution in the subtree generated from $\langle G_i'', k_i'' \rangle$, based on Algorithm B2. Processor P_i chooses a son of the node at random and expands it until a solution is found or the partial cover is larger than k. If a solution is not found, return to the subtree to get a still not explored son, until all the subtree is traversed. If a solution is encountered, the other processors are notified to interrupt.

In the algorithm of Cheetham *et al.* [4], the major part of the computational work occurs when each processor P_i, $0 \leq i < p$, computes locally the search tree from $\langle G''_i, k''_i \rangle$, where Algorithm B2 is used. As all the p subtrees are traversed simultaneously, it is possible that the parallel algorithm visits nodes that the sequential algorithm would not visit.

4 Implementation Details

In this section we present some implementation details of the parallel FPT algorithm and discuss the data structures utilized in our implementation. We use C/C++ and the MPI communication library.

The program receives as input a text file describing a graph G by its adjacency list and an integer k that determines the maximum size for the vertex cover desired. Let n be the number of vertices and m the number of edges of graph G and p the number of processors to run the program.

At the beginning of the reduction to problem kernel phase, the input adjacency list of graph G is transformed into a list of corresponding edges and distributed among the p processors. Each processor P_i, $0 \leq i < p$, receives m/p edges and is responsible for controlling the degrees of n/p vertices.

Each processor sorts the edges received by the identifier of the first vertex they are incident with, and obtains the degree of such vertices. Notice it is possible for a processor to compute the degree of the vertices that are of responsibility for another processor. In this case, the results are sent to the corresponding processor.

After this communication, the p processors can identify the local vertices with degree larger than k and send this information to the others, so that each processor can remove the local edges incident with these vertices. All the remaining edges after the removal, that form the new graph G', are sent to all the processors. In this way, at the end of this phase, each processor has the instance generated by the method of reduction to problem kernel and the partial cover (vertices of degree smaller than k), that is, the root of the bounded search tree. The p processors transform the list of edges corresponding to graph G' again into an adjacency list, that will be used in the next phase.

The resulting adjacency list from the reduction to problem kernel is implemented as a doubly linked list of vertices. Each node x of this list of vertices contains a pointer to a doubly linked list of pointers, whose elements represent all the edges incident with x, that we denote, for simplicity, by the list of edges of x. Each node of the list of edges of x points to the node of the list of vertices that contains the other extreme of the edge. In spite of the fact that graph is not a directed graph, each edge is represented twice in distinct lists of edges. Thus each node of the list of edges contains also a pointer to its other representation. In Fig. 2 we present an example of a graph and the data structure to store it.

The insertion of a new element in the list of vertices takes $O(n)$ time, since it is necessary to check if such elements already exist. In the list of edges, the insertion of a new element, in case it does not yet exist, results in the insertion of elements in the two lists of edges incident with its two extremes and also takes $O(n)$ time.

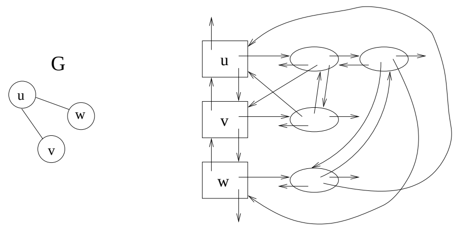

Fig. 2. The data structure used to store the graph G.

The removal of a vertex or an edge is a rearrangement of the pointers of previous and next elements of the list. They are not effectively deallocated from memory, they are only removed from the list. Notice that the edges incident with it are removed automatically with the vertex. However, we still have to remove the other representation. As each edge has a pointer to its other representation, we spend $O(1)$ time to remove it from the list of edges of the other vertex. Therefore, we spend $O(k)$ time to remove a vertex from the list, since the vertices of the graph have degree bounded by k. In our implementation, we store in memory only the data relative to the node of the bounded search tree being worked on.

Since we use depth first traversal in the bounded search tree, we need to store some information that enables us to go up the tree and recover a previous instance of the graph. Thus our program uses the *backtracking* technique. Such information is stored in a stack of pointers to removed vertices and edges. Adding an element in the stack takes $O(1)$ time. Removing an element from the stack and put it back in the graph also takes $O(1)$ time, since the removed vertex or edge has pointers to the previous and next elements in the list.

The partial vertex cover is also a stack of pointers to vertices known to be part of the cover. To add or remove an element from the cover takes $O(1)$ time.

At the beginning of the bounded search tree phase, all the p processors contain the instance ($\langle G', k' \rangle$) and the partial vertex cover resulting from the phase of reduction to problem kernel. As seen in Section 3.3, there exists a bounded tertiary complete search tree T with p leaf nodes. Each processor P_i, $0 \leq i < p$, uses Algorithm B1, generates the unique path in tree T from the root to the leaf node γ_i of tree T. Then, each processor P_i applies Algorithm B2 in the subtree whose root is the leaf node γ_i, until finding a solution or finishing the traversal.

In Algorithm B1, we search a path that starts at a vertex and passes through at most three edges. In our implementation, this initial vertex is always the first vertex of the list and, therefore, the same tree T is generated in all the executions of the program.

In the implementation of Algorithm B2, to obtain constant time for the selection of a vertex for the cases of this algorithm, we use 6 auxiliary lists of pointers to organize the vertices of the graph according to its degree (0, 1, 2, 3, 4 and 5 or more). Furthermore, each vertex of the graph also has a pointer to its representative in the list of degrees, therefore in any change of degree of a vertex implies $O(1)$ time to change it in the list of degrees.

5 Experimental Results

We present the experimental results by implementing the CGM algorithm of Cheetham *et al.* [4], using the data structures and the description of the previous section. Our parallel implementation will be called **Par-Impl**. Furthermore, we also implemented Algorithm B2 in C/C++, to be called **Seq-Impl**.

The computational environment is a Beowulf cluster of 64 Pentium III processors, with 500 MHz and 256 MB RAM memory each processor. All the nodes are interconnected by a Gigabit Ethernet switch. We used Linux Red Hat 7.3 with g++ 2.96 and MPI/LAM 6.5.6.

The sequential times were measured as wall clock times in seconds, including reading input data, data structures deallocation and writing output data. The parallel times were also measured as wall clock time between the start of the first processor and termination of the last process, including I/O operations and data structures deallocation.

In our experiments we used conflict graphs that were kindly provided by Professor Frank Dehne (Carleton University). These graphs represent sequences of amino acid collected from the NCBI database. They are Somatostatin, WW, Kinase, SH2 (src-homology domain 2) and PHD (pleckstrin homology domain). The Table 1 shows a summary of the characteristics of these graphs (name, number of vertices, number of edges, size of desired cover and size of the cover to search for after the reduction to problem kernel).

Graph	$\|V\|$	$\|E\|$	k	k'
Kinase	647	113122	495	391
PHD	670	147054	601	600
SH2	730	95463	461	397
Somatostatin	559	33652	272	254
WW	425	40182	322	318

Table 1. Sequences and corresponding graphs and cover sizes used in experiments.

In Fig. 3 we compare the times obtained by executing Seq-Impl and Par-Impl in a single processor (3 virtual processors) and Par-Impl in 27 processors. To run Par-Impl in a single processor we used MPI/LAM simulation mode, that simulates p virtual processors as independent processes on the same physical processor. The time obtained by Par-Impl in a single processor is the sum of the wall clock times of the individual processes plus the overhead created by their communication. The tests were carried out for the graphs PHD, Somatostatin and WW. These input data were chosen because their sequential times are reasonable. To obtain the averages, we ran Seq-Impl 10 times for each data set and

Par-Impl 30 times for each data set. In spite of the fact we are using a single processor to run the parallel implementation, the time was significantly much smaller. This is justified by the fact of having more initial distinct points in the bounded search tree, such that from one of them we can find a path that takes to the cover more quickly.

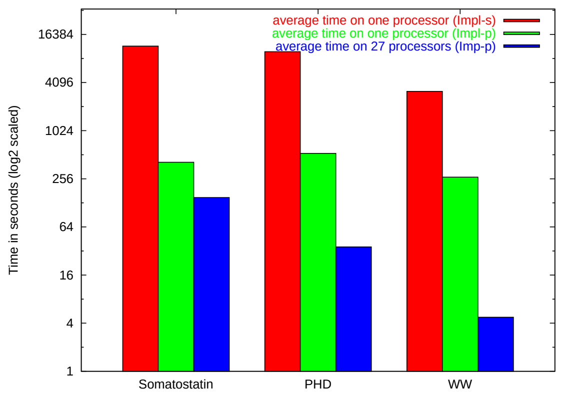

Fig. 3. Comparison of sequential and parallel times.

In Fig. 4 we show the average of the parallel times obtained in 27 processors. Our parallel implementation can solve problem instances of size $k \geq 400$ in less than 3 minutes. For example, graph PHD ($k = 601$) can be solved in less than 1 minute. Notice that k-Vertex Cover problem is considered well solved for instances of $k \leq 200$ by sequential FPT algorithms [7]. It is important to emphasize that the time of FPT algorithm grows exponentially in relation to k. Again we use 30 time samples to get the average time. Observe the times obtained and the Table 1. We see that the parallel wall clock times do not strictly increase with either k or k'. This makes us conclude that the graph structure is the responsible for the running time.

The parallel times, using 3, 9 and 27 processors for the graphs PHD, Somatostatin and WW are shown in Fig. 5. Notice the increase in the number of processors does not necessarily imply a greater improvement on the average time, in spite of the always observed time reduction. Nevertheless, the use of more processors increases the chance of determining the cover more quickly, since we start the tree search in more points. Furthermore, it seems that the number of tree nodes with a solution also has some influence on the running times. As we do the depth first traversal in the bounded search tree, a wrong choice of a son to visit means that we have to traverse all the subtree of the son before choosing another son to visit.

For the graphs PHD, SH2, Somatostatin and WW we could guarantee, in less than 75 minutes, the non existence of covers smaller than that determined by the parallel algorithm, confirming the minimality of the values obtained. For

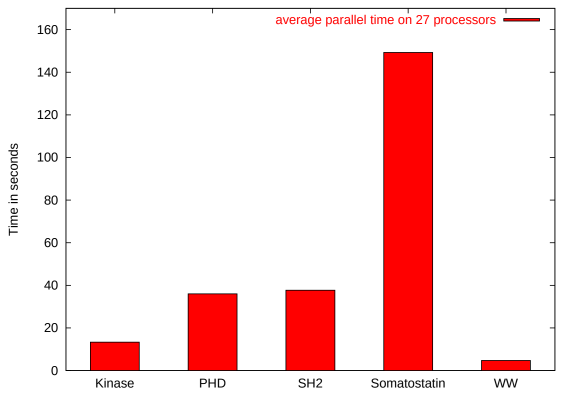

Fig. 4. Average wall clock times for the data sets on 27 real processors.

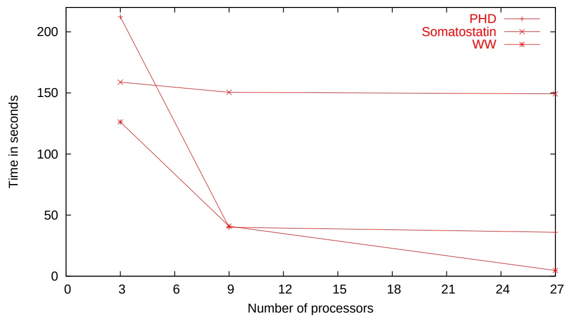

Fig. 5. Average wall clock times on 3, 9 and 27 processors for PHD, Somatostatin and WW.

this, all the possible nodes of the bounded search tree were generated. For the graph Kinase this was not possible in an acceptable time.

Our results were compared with those presented in Cheetham *et al.* [4], who used a Beowulf Cluster of 32 Xeon nodes of 1.8 GHz and 512 MB of RAM. All the nodes were interconnected by a Gigabit Ethernet switch. Every node was running Linux Red Hat 7.2 with gcc 2.95.3 and MPI/LAM 6.5.6.

Our experiments are very relevant, since we used a computational platform that is much inferior than that used in Cheetham *et al.* [4]. The parallel times obtained in our experiments were better. We considered that the choice of good data structures and use of the backtracking technique were essential to obtain our relevant results. For the graphs Kinase and SH2 we obtained parallel times that are much better, a reduction by a factor of approximately 115. The time for the graph PHD was around 16 times better. For the graphs Somatostatin and WW the times are slightly better. As we did not have access to the implementation of Cheetham *et al.* [4], we tested several data structures in our implementation. In the final version we used that implementation that gave the best performance, together with the backtracking technique. More details can be found in Hanashiro [11].

Furthermore, the size of the covers obtained were smaller for the following graphs: Kinase (from 497 to 495), PHD (from 603 to 601) and Somatostatin (from 273 to 272). It is important to emphasize that the reduction in the size of the cover implies the reduction on the universe of existing solutions in the bounded search tree, which in turn gives rise to an increase in the running time.

6 Conclusion

FPT algorithms constitute an alternative approach to solve NP-complete problems for which it is possible to fix a parameter that is responsible for the combinatorial explosion. The use of parallelism improve significantly the running time of the FPT algorithms, as in the case of the k-Vertex Cover problem.

In the implementation of the presented CGM algorithm, the choice of the data structures and the use of the backtracking technique were essential to obtain the relevant experimental results. During the program design, we utilized several alternative data structures and their results were compared with those of Cheetham *et al.* [4]. Then we chose the design that obtained the best performance. Unfortunately we did not have access to the implementation of Cheetham *et al.* to compare it with our code.

We obtained great improvements on the running times as compared to those of Cheetham *et al.* [4]. This is more significant if we take into account the fact that we used an inferior computational environment. Furthermore, we improved the values for the minimum cover and guaranteed the minimality for some of the graphs.

The speedups of our implementation with that of Cheetham *et al.* [4] vary very much. The probable cause of this may lie in the structures of the input graphs, and also in the number of solutions and how these solutions are distributed among the nodes of the bounded search tree.

For the input used, only for the Thrombin graph we did not obtain better average times, as compared to those of Cheetham *et al.* [4]. To improve the results, we experimented two other implementations, by introducing randomness in some of the choices. With these modifications, in more experiments we get lower times for the Thrombin graph, though we did not improve the average. For some of the graphs, the modification increases the times obtained, and does not justify its usage.

Acknowledgments

The authors would like to thank Prof. Frank Dehne (Carleton University) who kindly provided us the conflict graphs, Prof. Edson N. Cáceres (UFMS) for his assistance, the Institute of Computing/UNICAMP for giving the permission to use the machines, and finally the referees for their helpful comments.

References

1. Downey, R.G., Fellows, M.R.: Parameterized Complexity. Springer-Verlag (1998)
2. Balasubramanian, R., Fellows, M.R., Raman, V.: An improved fixed-parameter algorithm for vertex cover. Information Processing Letters **65** (1998) 163–168
3. Dehne, F., Fabri, A., Rau-Chaplin, A.: Scalable parallel computational geometry for coarse grained multicomputers. In: Proceedings of the ACM 9th Annual Computational Geometry. (1993) 298–307
4. Cheetham, J., Dehne, F., Rau-Chaplin, A., Stege, U., Taillon, P.: Solving large FPT problems on coarse grained parallel machines. Journal of Computer and System Sciences **4** (2003) 691–706
5. Downey, R.G., Fellows, M.R.: Fixed-parameter tractability and completeness I: Basic results. SIAM J. Comput. **24** (1995) 873–921
6. Downey, R.G., Fellows, M.R.: Parameterized complexity after (almost) 10 years: Review and open questions. In: Combinatorics, Computation & Logic, DMTCS'99 and CATS'99. Volume 21, número 3., Australian Comput. Sc. Comm., Springer-Verlag (1999) 1–33
7. Downey, R.G., Fellows, M.R., Stege, U.: Parameterized complexity: A framework for systematically confronting computational intractability. In: Contemporary Trends in Discrete Mathematics: From DIMACS and DIMATIA to the Future. Volume 49 of AMS-DIMACS Proceedings Series. (1999) 49–99
8. Niedermeier, R.: Some prospects for efficient fixed parameter algorithms. In: Conf. on Current Trends in Theory and Practice of Informatics. (1998) 168–185
9. Buss, J.F., Goldsmith, J.: Nondeterminism within P. SIAM J. Comput. **22** (1993) 560–572
10. Chan, A., Dehne, F.: A note on course grained parallel integer sorting. In: 13th Annual Int. Symposium on High Performance Computers. (1999) 261–267
11. Hanashiro, E.J.: O problema da k-Cobertura por Vértices: uma implementação FPT no modelo CGM. Master's thesis, Universidade Federal de Mato Grosso do Sul (2004)