A Parallel FPT Application For Clusters *

James Cheetham
Insitute of Biochemistry
Carleton University
Ottawa, Canada
jcheetha@ccs.carleton.ca

Frank Dehne
School of Computer Science
Carleton University
Ottawa, Canada
frank@dehne.net

Andrew Rau-Chaplin
Faculty of Computer Science
Dalhousie University
Halifax, Canada
arc@cs.dal.ca

Ulrike Stege
Dept. of Computer Science
University of Victoria
Victoria, Canada
stege@csr.uvic.ca

Peter J. Taillon
School of Computer Science
Carleton University
Ottawa, Canada
ptaillon@scs.carleton.ca

Abstract

Fixed-parameter tractability (FPT) techniques have recently been successful in solving NP-complete problem instances of practical importance which were too large to be solved with previous methods. In this paper we show how to enhance this approach through the addition of parallelism, thereby allowing even larger problem instances to be solved in practice. More precisely, we demonstrate the potential of parallelism when applied to the bounded-tree search phase of FPT algorithms. We apply our methodology to the k-VERTEX COVER problem which has important applications, e.g., in multiple sequence alignments for computational biochemistry. We have implemented our parallel FPT method for the k-VERTEX COVER problem using C and the MPI communication library, and tested it on a PC cluster. This is the first experimental examination of parallel FPT techniques. We have tested our parallel k-VERTEX COVER method on protein sequences obtained from the National Center for Biotechnology Information. As part of our experiments, we solved larger instances of k-VERTEX COVER than in any previously reported implementations. For example, our code can solve problem instances with $k \geq 400$ in less than 1.5 hours. Since our parallel FPT algorithm requires only very little communication between processors, we expect our method to also perform well on Grids.

Key Words: *Fixed-Parameter Tractability, k-Vertex Cover, Computational Biochemistry.*

1. Introduction

NP-complete problems abound in many important application areas ranging from computational biology to network optimization. For scientists and engineers with computational problems, merely learning that their problems are NP-complete does not satisfy their need to solve these problems for the instances at hand. Fixed-parameter tractability (FPT) is a new technique for confronting the obstacle of NP-Completeness [9, 10, 11, 12, 13, 14]. FPT algorithms have been successful in solving NP-complete problem instances of practical importance which were too large to be solved with previous methods [9]. Most FPT algorithms consist of two phases: *kernelization* where the problem is reduced to a much smaller instance and *bounded-tree search* where the problem is solved on the smaller instance through the traversal of a search tree. The Computational Biochemistry Research Group at the ETH Zürich has successfully incorporated the FPT approach for VERTEX COVER problems arising in multiple sequence alignments for computational biochemistry research [18, 22, 25]. In this paper, we further increase the size of problems that can be solved via FPT methods by showing how the FPT approach can be effectively parallelized on a cluster. We have implemented a parallel FPT method for the k-VERTEX COVER problem using C and the MPI communication library, and tested it on a PC cluster. This is the first experimental examination of parallel FPT techniques. Our experiments, presented in Section 4, study the speedup and scalability of our method on conflict graphs for gene sequences obtained from the National Center for Biotechnology Information. For scientists and engineers who have NP-complete problems to solve, the real test for any new method is how large a "real-world" problem instance it can solve. In [9], the authors consider the k-VERTEX COVER problem solvable for $k \leq 200$ within an 8 hour running time. In contrast, our parallel code is able to solve instances of the k-VERTEX COVER problem with $k \geq 400$ in less than 1.5 hours, using 27 processors. This is a significant improvement since the time of sequential FPT algorithms grows exponentially in k. We

Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'03)
0--7695-1919-9/03 $17.00 © 2003 IEEE

are currently in the process of building a web portal where Biochemists can submit gene sequences, have the conflict graph computed, and then have the conflict graph processed by our parallel k-VERTEX COVER algorithm.

Another important aspect of our parallel k-VERTEX COVER method is that it requires only a very small amount of communication between processors. Hence, we expect our method to also perform well on Grids. Michael A. Langston and his group at the University of Tennessee are currently in the process of porting our method to a Grid environment [19].

This paper does, in fact, present a *general methodology* for parallelizing the bounded-tree search phase of FPT algorithms. For ease of presentation, we introduce our tree search parallelization method by describing immediately its application to the k-VERTEX COVER problem. The generalization to parallel tree search for other FPT algorithms is straight-forward.

Parallel FPT algorithms have previously been presented in [2, 6] but without experimental performance analysis. These methods parallelize only the kernelization phase of the respective sequential FPT algorithm and leave the tree search unchanged. The parallel methods in [2, 6] are shown to be in NC (for fixed parameter k) which is the main goal of those papers. Unfortunately, these results are of little use in practice. Typical FPT implementations spend minutes on the kernelization and hours on the tree search. Hence, it is important to parallelize *both* phases to obtain any speedup in practice. The main contribution of this paper is to provide an efficient implementation of parallel bounded-tree search on a PC cluster.

The remainder of this paper is organized as follows. Section 2 reviews the definition of fixed parameter tractability and the k-VERTEX COVER problem. In Section 3, we present our main result, a parallel FPT algorithm for the k-VERTEX COVER problem. Section 4 presents the experimental performance results for our method on a PC cluster, and Section 5 concludes the paper.

2. Review: Fixed-Parameter Tractability and the k-VERTEX COVER Problem

Fixed-parameter tractability (FPT) has been proposedstudied in [1, 9, 10, 11, 12, 13, 14] as a means of confronting the obstacle of NP-Completeness. Let Σ be a finite alphabet and let L be a parameterized problem such that $L \subseteq \Sigma^* \times \Sigma^*$. Problem L is *fixed-parameter tractable*, or FPT, if there exists an algorithm that decides, given an input $(x, y) \in \Sigma^* \times \Sigma^*$, whether $(x, y) \in L$, in time $f(k) + n^\alpha$, where $|x| = n$, $|y| = k$ is a parameter, α is a constant independent of n and k, and f is an arbitrary function. The goal is to isolate, in the parameter k, the component of the input that causes the exponential time. The two fundamental algorithmic techniques for solving FPT problems are *kernelization* and *bounded-tree search* [10]. As a two phase approach, kernelization and bounded-tree search form the basis of many FPT algorithms. The first phase, kernelization, reduces the problem, in polynomial time, to another problem instance bounded in size by a function of k. It was shown in [9] that a problem is in FPT if and only if it is kernelizable. The second phase, bounded-tree search, then attempts to solve the latter problem by exhaustive search, requiring time exponential in k.

The k-VERTEX COVER problem has important applications in multiple sequence alignments for computational biochemistry [25]. The VERTEX COVER problem is defined as follows [17]: given a graph, $G = (V, E)$, determine a set, $VC \subseteq V$, containing a minimum number of vertices such that for all $(x, y) \in E$, either $x \in VC$ or $y \in VC$. The k-VERTEX COVER problem consists of finding a VERTEX COVER of size k. In multiple alignments between gene sequences, whenever there are conflicts between sequences, a way to resolve these conflicts is to exclude some sequences from the sample. Define a conflict graph as a graph where every sequence is a vertex and every edge is a conflict between two sequences. A conflict may be defined when the alignment of these two sequences has a very poor score. The goal is to remove the fewest possible sequences that will eliminate all conflicts, which is equivalent to finding a minimum VERTEX COVER for the conflict graph.

The VERTEX COVER problem is known to be NP-Complete, but in the context of parameterized complexity the problem is fixed-parameter tractable [3, 5, 9, 10, 11, 12, 13, 14, 15, 20, 24, 25]. The current theoretically best sequential FPT algorithm for the k-VERTEX COVER Problem has a time complexity of $O(kn + 1.271^k k^2)$ [5]

3. Parallel Kernelization and Bounded-Tree Search for the k-VERTEX COVER Problem

This paper describes an efficient parallelization of both, kernelization and bounded-tree search for FPT methods. In this section, we describe our general methodology using the example of the k-VERTEX COVER problem. For a list of other FPT problems that can be solved via kernelization and bounded-tree search see [10]. We first present a brief overview of our parallel k-VERTEX COVER algorithm, with details to follow in Sections 3.1 and 3.2.

Our parallel FPT method assumes a set of p processors, P_0, P_1, ..., P_{p-1} where each processor has $O(N/p)$ local memory. Parameter N refers to the total problem size. Our parallel method uses, as a building block, portions of two sequential FPT algorithms described in [1]. The first algorithm in [1] combines Buss' kernelization technique [3] with a 3-level, depth-first search strategy that produces a 3-ary search tree (referred to as *THEOREM 1* in [1]). The second algorithm in [1] combines Buss' kernelization technique with case-based reduction rule application to determine a k-VERTEX COVER (referred to as *THEOREM 2* in

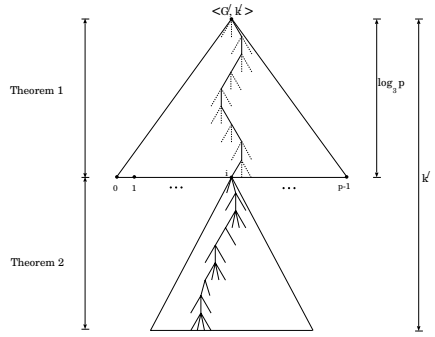

Figure 1. Search Path For Processor P_i in Algorithm 2, Using *THEOREM 1* And *THEOREM 2* of [1].

[1]). Our parallel method first performs a parallel kernelization on the problem instance $\langle G = (V, E), k\rangle$, with the resulting instance $\langle G', k'\rangle$ then being broadcast to all processors. This phase, which is outlined in Sections 3.1, is straight-forward. Let VC_{kern} be the set of vertices determined by the kernelization phase to be in the vertex cover set, VC. Sections 3.2 then describes our parallel bounded-tree search method. Each processor, P_i, $0 \leq i \leq p-1$, locally and deterministically executes the search tree phase of the *THEOREM 1* algorithm on its instance of $\langle G', k'\rangle$ as follows: P_i selects exactly the branching nodes that lead it to leaf i at depth $\log_3 p$ of the search tree. This approach is similar to search-frontier splitting, as each processor now has a unique problem instance, $\langle G''_i, k''_i\rangle$. Each processor, P_i, $0 \leq i \leq p-1$, then locally performs a randomized depth-first search of the subtree rooted at leaf i, starting with instance $\langle G''_i, k''_i\rangle$. See Figure 1. When a processor finds a solution, VC_i, it outputs the set $VC_{kern} \bigcup VC_i$ and signals all other processors to terminate.

In the following two sections we describe in detail our parallelization of the kernelization and the tree search, respectively.

3.1. Parallel Kernelization

The parallelization of the kernelization phase is straight-forward. For a graph $G = (V, E)$ and parameter k, Buss' kernelization algorithm consists of the following steps: find the set S consisting of all vertices v such that $deg(v) > k$. Let $|S| = b$. If $b > k$ then we conclude that there can be no k-sized vertex cover in G. Otherwise, include S in the vertex cover, remove all the elements of S from V.[1] Let $k' = k - b$. If the resulting graph, G', has more than $k \cdot k'$ edges, then we can conclude no k-sized cover is possible. Otherwise, $\langle G', k'\rangle$ is a kernelized instance of $\langle G, k\rangle$.

[1]For the remainder, we assume that whenever a vertex v is removed from a graph, all edges adjacent to v are removed as well.

In the parallel setting, this operation reduces to a constant number of parallel integer sorts where edges are sorted by vertex id in order to identify the vertices with $deg(v) > k$. This sort can be implemented via deterministic sample sort [4]. Note that other kernelization rules can be applied as described in [9] and [1]. These rules can also be reduced to a constant number of parallel integer sorts.

Algorithm 1 *Parallel Kernelization*
Input: $\langle G = (V, E), k\rangle$. **Output:** $\langle G', k'\rangle$ or *"No"*.
(1.1) Simulate Buss' kernelization algorithm on $G = (V, E)$ via $O(1)$ parallel integer sorts, using deterministic integer sample sort [4].
(1.2) Output either a kernelized graph $\langle G' = (V', E'), k'\rangle$, or VC ($\leq k$), or *"No"*.
— End of Algorithm —

Algorithm 1 performs kernelization with local computation time $O(\frac{kn}{p})$ and only a constant number of h-relation (MPI_AllToAllv) operations for communication between processors.

3.2. Parallel Bounded-Tree Search

We first recall a few facts about sequential bounded-tree search. Let $\langle G'' = (V'', E''), k''\rangle$ be a problem instance associated with a search tree node x currently under consideration in the bounded-tree search and let VC be the current set of vertices known to be in the vertex cover. The algorithm described by *THEOREM 1* of [1] consists of repeating the following steps until either the correct VC is found, or it is determined that G does not have a k-cover. **Step 1:** Randomly select a vertex, $v \in V''$. **Step 2:** Starting from v, perform a depth-first search traversing at most three edges. **Step 3:** Based on the possible paths derived from the search in Step 2, either expand node x into three children (Cases 1 and 2) or process immediately (Cases 3 and 4):
Case 1. The path obtained in Step 2 is a simple path of length 3 consisting of a sequence of vertices v, v_1, v_2, v_3. Associate three children (i.e., subproblems) with node x as follows:
- $\langle G''' = (V'' - \{v, v_2\}, E'''), k''' = k'' - 2\rangle$; $VC = VC \bigcup \{v, v_2\}$
- $\langle G''' = (V'' - \{v_1, v_2\}, E'''), k''' = k'' - 2\rangle$; $VC = VC \bigcup \{v_1, v_2\}$
- $\langle G''' = (V'' - \{v_1, v_3\}, E'''), k''' = k'' - 2\rangle$; $VC = VC \bigcup \{v_1, v_3\}$

Case 2. The path obtained in Step 2 is a 3-cycle consisting of the following sequence of vertices v, v_1, v_2, v. Associate three children with node x as follows:
- $\langle G''' = (\ V'' - \{v, v_1\}, E'''), k''' = k'' - 2\rangle$; $VC = VC \bigcup \{v, v_1\}$
- $\langle G''' = (\ V'' - \{v_1, v_2\}, E'''), k''' = k'' - 2\rangle$; $VC = VC \bigcup \{v_1, v_2\}$

• $\langle G''' = (\ V'' - \{v, v_2\}, E'''), k''' = k'' - 2\rangle$; $VC = VC \bigcup \{v, v_2\}$

Case 3. The path obtained in Step 2 is a simple path of length 2 (i.e., pendant edge) consisting of a sequence of vertices v, v_1, v_2. This can be processed immediately as follows: $\langle G''' = (\ V'' - \{v_1, v_2\}, E'''), k''' = k'' - 1\rangle$; $VC = VC \bigcup \{v_1\}$.

Case 4. The path obtained in Step 2 is a simple path of length 1 (i.e., pendant edge) consisting of a sequence of vertices v, v_1. This can be processed immediately as follows: $\langle G''' = (\ V'' - \{v, v_1\}, E'''), k''' = k'' - 1\rangle$; $VC = VC \bigcup \{v\}$.

The sequential time complexity of the algorithm is $O((\sqrt{3})^k k^2 + kn)$. The algorithm described by *THEOREM 2* in [1] consists of scanning the adjacency list associated with a graph instance at a given search tree node for specific branching cases. See [1] for details regarding these reduction rules. Note that, this algorithm no longer guarantees a 3-ary search tree. The number of children created can be 2, 3, or 4, while the parameter (k) can decrease by as much as 8, depending on the rule that is applied. The sequential time complexity of the algorithm is $O((1.324718)^k k^2 + kn)$.

Our basic approach for parallelizing the tree search is as follows. We initially create the first $O(\log p)$ levels of the search tree in breadth-first fashion until we have obtained a search tree with p leaves. This is done using the algorithm described by *THEOREM 1*, in a deterministic fashion. We then assign each of the p leaves to one processor and let each processor continue searching the tree from its respective leaf. In this step, we use the algorithm described by *THEOREM 2*. We assure that this part of the tree search is randomized: that is, when a processor proceeds downwards in the search tree, it selects a random node among the still unexplored children. See Figure 1 for an illustration. The following describes our tree search parallelization in more detail.

Algorithm 2 *Parallel Tree Search*
Input: $\langle G', k'\rangle$. **Output:** VC ($\leq k$), or *"No"*.

(2.1) Consider the search tree T obtained by starting with graph G' and iteratively expanding the combinatorial search tree in breadth-first fashion, using the *THEOREM 1* algorithm, until there are exactly p leaves $\gamma_1 \ldots \gamma_p$. Every processor, P_i, $0 \leq i \leq p-1$, computes the unique path in T from the root to leaf γ_i. Let (G_i'', k_i''), $0 \leq i \leq p-1$, be the subgraphs and updated parameters associated with γ_i.

(2.2) Processor P_i, $0 \leq i \leq p-1$, starts with (G_i'', k_i'') and expands/searches the subtree below γ_i in a randomized, depth-first fashion, using reduction rules of the *THEOREM 2* algorithm, as follows:

Processor P_i randomly selects and expands one of the children, repeating this recursively until either a solution is found or the parameter is exhausted (i.e., there is no solution). P_i then backtracks in its subtree and randomly chooses another unexplored child. This process is repeated until a solution is found (in which case it notifies all other processors to halt) or the processor's subtree has been completely searched.

— End of Algorithm —

While the above algorithm is fairly simple, it is non-trivial to analyze its performance. Consider the path Λ in which a sequential algorithm traverses the search tree. The sequential processing time is determined by the number l_{seq} of nodes in Λ which need to be traversed until a first solution is found. The parallel algorithm essentially sets p equally spaced starting points on Λ and starts p search processes, one at each starting point. Let Λ_i be the portion of Λ assigned to processor P_i, and let l_i be the number of nodes in Λ which processor P_i needs to traverse until it finds a first solution. The parallel time is determined by $l_{par} = \min_{0 \leq i \leq p-1} l_i$, the minimum number of nodes that a process has to traverse until it reaches a solution node. The possible speedup observed corresponds to the ratio between l_{seq} and l_{par}. What speedup is obtained through this parallel exploration of subtrees? After all, only one solution needs to be found. Clearly, it is possible that the parallel algorithm examines many nodes that the sequential algorithm would never reach. The worst case occurs when $l_0 = \min_{0 \leq i \leq p-1} l_i$ and we obtain no speedup at all. What kind of speedup can we expect in the average case? The main problem is that the distribution of the solutions within the search tree is unknown [16].

A "balls-in-bins" model was used to predict the speedup that could be expected for our parallel tree search algorithm. Consider p processors and a path Λ of length L in which a sequential algorithm traverses the search tree. As a working hypothesis, assume that there are m solutions in the search tree which are randomly distributed (with uniform distribution) over the search path Λ. Consider an array of p rows and $n = L/p$ columns. The ith row corresponds to Λ_i and the entire array corresponds to Λ. Mark m random array elements as solutions and measure l_{seq} and l_{par} = $\min_{0 \leq i \leq p-1} l_i$.

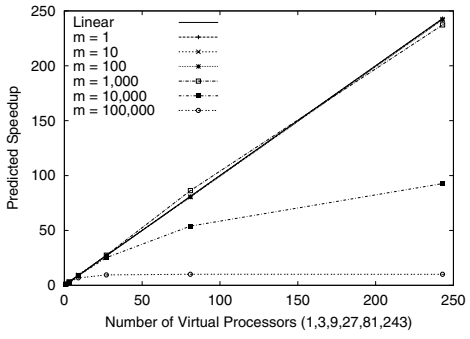

Figure 2. Simulated relative speedup estimation through "Balls in Bins" experiment.

The results of a simulation of the above "balls-in-bins" model are shown in Figure 2. Experiments were performed for $L = 1000000$, m = 1, 10, 100, 1000, 10000, 100000 and $p = 3, 9, 27, 81, 243$ processors. The x-axis represents the number p of processors and the y-axis represents the speedup $s_p = l_{seq}/l_{par}$. Each data point shown corresponds to the average of 150 experiments. The diagonal line, $s_p = p$ represents (optimal) linear speedup. Note that points above the diagonal are due to statistical error. The most striking result of the experiments is how close all data points are to the diagonal line for m = 1, 10, 100, 1000. These are the most realistic cases in practice because the number of actual k-VERTEX COVER solutions is small compared to the very large, exponential-size, search space. Even for m = 10000, that is where 1% of the entire search space correspond to solutions, we observe a speedup of about $p/2$. Only for m = 100000, that is where 10% of the entire search space correspond to solutions, we observe very low speedup. Note that in this case, any sequential method would find a solution in such a short time that parallelization is not even interesting. We ran the experiment for many other combinations of L, m, and p, and the results were always very similar.

The close to linear speedup for low density m/L, under uniform distribution assumption, can be explained as follows [8]. The expected number of nodes in Λ that need to be traversed by the sequential algorithm is given by $E(l_{seq}) = \frac{L}{m+1}$. The expected number of nodes $l_{par} = \min_{1 \leq i \leq p} l_i$ that need to be traversed by the parallel algorithm is bounded by $E(l_{par}) \leq \frac{L/p}{m+1} + p$. Therefore, we obtain speedup

$$E(s_p) \geq \frac{1}{\frac{1}{p} + \frac{m+1}{L/p}}$$

For $m \ll L/p$ the second part of the denominator becomes negligible and we get an expected speedup $E(s_p)$ of approximately p. This is what we observed in Figure 2 for $m \leq 1000$. It is important to note that the above inequality is only a coarse lower bound. The actual speedup can be considerably better. Furthermore, as the discussion in [23] suggests, the uniform distribution of the m solutions over the array examined above does *not* constitute a *good* scenario. On the contrary, when solutions are non-uniformly distributed, the processor whose search path starts close to a cluster has a high probability of finding a solution much faster than in the uniform case. Therefore it can be expected that the speedup observed is better in the non-uniform case than in the uniform case. For bounded-tree search for the k-VERTEX COVER and other FPT algorithms, one can usually assume that the distribution of solutions within the search tree is not uniform. In fact, this is what we observe in our experimental results presented in the next section.

4. Performance Analysis

In this section we discuss the experimental examination of our parallel FPT technique. We first discuss our setup and methodology as well as the data sets used for the evaluation. We then present the performance results obtained.

4.1. Experimental Setup and Methodology

We first implemented in C the sequential FPT algorithm described in [1] (*THEOREM 2*). We will refer to this sequential C code as `Code-s`. While recent theoretical improvements of the core result in [1], namely [5, 20], exhibit tradeoffs between small differences in the asymptotic running time and leading constants, we believe that execution times measured on a well crafted implementation of [1] are a good representation of the current sequential state-of-the-art.

We then implemented our parallel FPT method described in Section 3, using C and the MPI communication library, by adding the relevant C and MPI code to `Code-s`. We will refer to this parallel C/MPI code as `Code-p`. Note that, `Code-s` is the same as a one processor version of `Code-p` with all MPI calls disabled and all code removed that is not required for the one processor case.

For our experiments, we used a 32 node PC cluster with 1.8 GHz Intel Xeon processors, 512 MB RAM per node and 60 GB of disk storage per node. Every node was running Linux Redhat 7.2 with gcc 2.95.3 and MPI/LAM 6.5.6. The nodes were interconnected via a Cisco gigabit Ethernet switch. However, given the low bandwidth requirements of our method, a Fast Ethernet switch would have more than sufficed.

All sequential times were measured as wall clock times in seconds. All parallel times were measured as the wall clock time between the start of the first process and the termination of the last process. We will refer to the latter as *parallel wall clock time*. All times include the time taken to read the input graph from a file and write the solution into a file. Furthermore, all wall clock times were measured with no other user except us on the parallel machine. Our experiments proceeded in the following steps.

(1) Sequential Experiments.

(1a) Sequential `Code-s`: We executed `Code-s` on a single processor of our processor cluster and measured the sequential wall clock time. **(1b) Sequential `Code-p`:** We executed `Code-p` on a single processor of our processor cluster, using multiple virtual processors (i.e. MPI/LAM processes), and measured the sequential wall clock time.

(2) Parallel Experiments.

(2a) `Code-p` Parallel Wall Clock Times: We executed `Code-p` on our processor cluster and measured the parallel wall clock time. **(2b) `Code-p` Relative Speedup:** We executed `Code-p` on our processor cluster and measured the relative speedup with respect to parallel wall clock time, where the "baseline" (i.e.

time for one processor) was set to the minimum of the sequential times measured in Steps 1a and 1b.

We briefly discuss why Step 1b is required. When executing `Code-p` on a single processor using multiple virtual processors (`Code-p`/simulated), we observed problem instances where `Code-p`/simulated is faster than `Code-s`. This is a very interesting aspect of our parallel k-VERTEX COVER method. Even on a single processor, `Code-p`/simulated starts multiple concurrent searches within the search tree. If one of those searches starts at a point close to an actual solution then `Code-p`/simulated can outperform `Code-s`. Therefore, for the calculation of the speedup in Step 2b, the minimum of the sequential times measured in Steps 1a and 1b must be compared with the parallel time obtained by `Code-p` on multiple processors.

4.2. Relative Speedup Measurements

For our speedup experiments we used conflict graphs derived for gene sequences obtained from the National Center for Biotechnology Information (NCBI), http://www.ncbi.nlm.nih.gov/. Protein modules that comprise large families of organisms were chosen. Proteins that are closely related have more similar amino acid sequences for an orthologous protein than more distantly related proteins. This information can be used to construct phylogenetic trees which represent relatedness of proteins. For our speedup experiments, we selected two sets of sequences: Somatostatin and WW. Somatostatin is a neuropeptide involved in the regulation of many functions in different organ systems. WW is a small protein domain that binds proline rich sequences in other proteins and is involved in cellular signaling.

The sequences in each data set were aligned using ClustalW [26], a hierachical multiple alignment program that generates pairwise alignments for all of the input sequences and then ranks the scores of the pairwise alignments. The conflict graph, i.e. the input for the k-VERTEX COVER problem, was created by selecting all sequences in the data set as vertices and selecting all edges between sequences whose alignment had a score below a given threshold Δ. The Δ values used are shown in Table 1, together with the sizes of the resulting conflict graphs and the values of k and k'.

Data Set	Δ	$\|V\|$	$\|E\|$	k	k'
Somatostatin	10	559	33652	273	255
WW	10	425	40182	322	318

Table 1. Sequences Used And Resulting Graph Sizes.

To explore general graph classes and known hard problem instances, we also tested our parallel FPT method on random graphs and grid graphs (see Table 2). We show results for one random graph and one grid graph which are typical for the results obtained in our experiments for these classes of graphs.

	$\|V\|$	$\|E\|$	$k = \|VC\|$	k'
Random	220	2155	122	122
Grid	289	544	145	145

Table 2. Random and Grid Graphs Used.

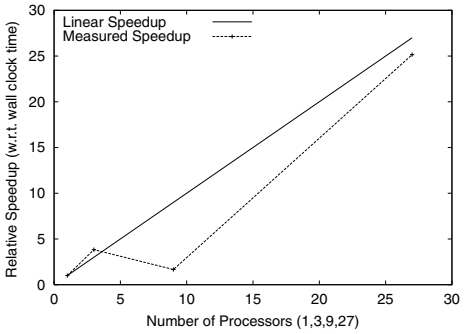

Figure 3. Average Relative Speedup for Somatostatin.

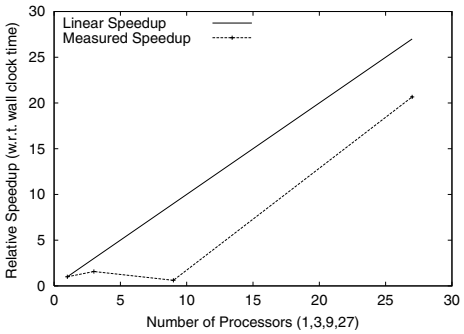

Figure 4. Average Relative Speedup for WW.

Figures 3 and 4 show the relative speedups measured for the Somatostatin and WW data sets and Figures 5 and 6 shows the relative speedups measured for the random and grid graphs. Each data point represents the average of 20 experiments. For Figures 3 and 4, we observe that the average relative speedup does not grow monotonically. For 27 processors, the average relative speedup is larger than 20. For a smaller number of processors we observed some "noise" in the average relative speedup caused by considerable variations in individual running times. Some "lucky draw"

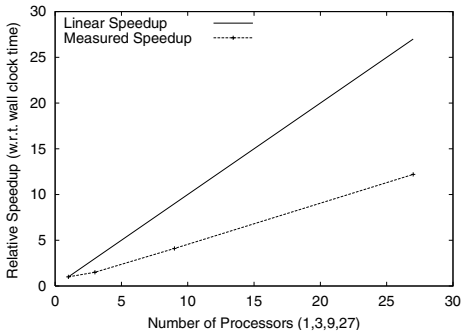

Figure 5. Average Relative Speedup for a Random Graph

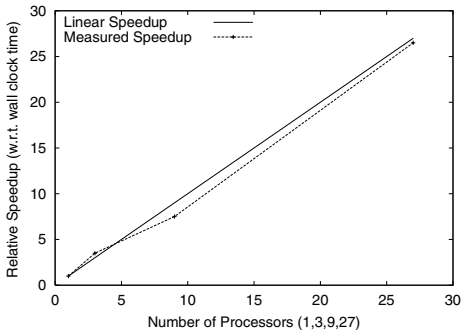

Figure 6. Average Relative Speedup for a Grid Graph

or "bad luck" events can occur where the search happens to find a solution near instantaneously or starts from points that are all very far away from solutions. With more starting points, i.e. more processors, the situation appears to become more stable.

For Figures 5 and 6, we observe that the average relative speedup grows monotonically in both cases. For the random graph data set in Figure 5, the slope of the average relative speedup curve is considerably lower. We observed the same effect for other random graphs. For the grid graph data set in Figure 6, we note that there exist exactly two solutions. As discussed in Section 3.2 and illustrated in Figure 2, the number of solutions in the search tree is very important for the relative speedup. For the grid graph, the number of possible solutions is very small. We conjecture that this is the reason why, in Figure 6, the slope of the average relative speedup curve is close to linear.

4.3. "Pushing The Envelope"

These experiments consisted of solving, on our PC cluster, individual problems for large values of k and k'.

Data Set	Δ	$\|V\|$	$\|E\|$	k	k'
Thrombin	15	646	62731	413	413
SH2	10	730	95463	461	397
Kinase	16	647	113122	497	397
PHD	10	670	147054	603	603

Table 3. Sequences Used And Resulting Graph Sizes

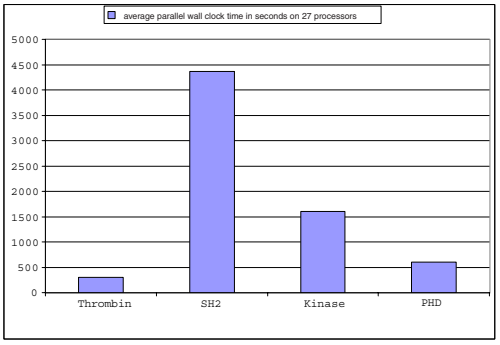

Figure 7. Average Parallel Wall Clock Times

We processed the following amino acid sequences obtained from the NCBI: Thrombin, SH2, Kinase, and PHD. Thrombin is a protease involved in the blood coagulation cascade and promotes blood clotting by converting fibrinogen to fibrin. SH2 (src-homology 2) domain protein modules are involved in targeting proteins to specific sites in cells by binding to phosphortyrosine. Protein kinases comprise a large and important family of enzymes involved in cellular regulation. PHD (pleckstrin homology domain) is a protein domain about 100 amino acid residues in length that is involved in cellular signaling. The sequences in each data set were again aligned using ClustalW, and conflict graphs were obtained using the threshold Δ values shown in Table 3. Observe that the conflict graphs have large values of k within the 400 to 600 range. Furthermore, k' is very close to k, indicating that these are hard instances.

We executed `Code-p` on 27 processors of our PC cluster and measured the average parallel wall clock time. Figure 7 shows the results. Each data point represents the average of ten experiments. We solved larger instances of k-VERTEX COVER than in any previously reported implementation. Our parallel FPT method is able to solve "real world" problem instances of size

$k \geq 400$ in less than 1.5 hours, whereas previously, for sequential FPT methods, only k-VERTEX COVER problems for $k \leq 200$ were considered solvable [9]. This is a significant improvement since the time of FPT algorithms for the k-VERTEX COVER problem grows exponentially in k.

5. Conclusion

In this paper, we have studied the potential of parallelism when applied to the bounded-tree search phase of FPT algorithms. We have designed and implemented a new parallel FPT method for the k-VERTEX COVER problem and tested it on a PC cluster, thereby providing the first experimental examination of parallel FPT techniques. By solving, e.g., problem instances with $k \geq 400$ in less than 1.5 hours, our code can handle larger instances of k-VERTEX COVER than any previously reported implementation.

Our parallel FPT algorithm requires only a very low amount of communication between processors. Hence, we expect our method to also perform well on Grids. Michael A. Langston and his group at the University of Tennessee are currently in the process of porting our method to a Grid environment [19].

The High Performance Computing Virtual Laboratory (HPCVL, http://www.hpcvl.org), which has been providing the PC cluster for our experiments, has expressed interest in providing a "parallel FPT tool" for Biochemists. We are currently in the process of building a Web portal where Biochemists can submit gene sequences, have them aligned by CLUSTAL, and then have our parallel k-VERTEX COVER algorithm applied to the conflict graph. Surprisingly, for large numbers of gene sequences, CLUSTAL has become a bottleneck. To rectify this, we are currently also in the process of parallelizing CLUSTAL for use in our web portal. The combined solution will allow Biochemists to analyze very large sets of gene sequences via our HPCVL portal.

References

[1] R. Balasubramanian, M.R. Fellows, V. Raman. "An Improved Fixed-Parameter Algorithm for Vertex Cover". *IPL*, Vol.65, 163–168, 1998.

[2] H.L. Bodlaender, R.G. Downey, M.R. Fellows. "Appl. of Parameterized Compl. to Problems of Parallel and Distr. Comput." Unpubl. abstract, 1994.

[3] J.F. Buss, J. Goldsmith. "Nondeterminism within P". *SIAM J. Computing*, Vol.22, 560–572, 1993.

[4] A. Chan, F. Dehne. "A Note on Coarse Grained Parallel Integer Sorting". *Proc. HPCS*, 261–267, 1999.

[5] J. Chen, I.A. Kanj, W. Jia. "Vertex Cover: Further observations and further improvements". *Proc. WG'99*, LNCS, 1999.

[6] M. Cesati, M. Di Ianni. "Parameterized Parallel Complexity". In *Proceedings of the 4th International Euro-Par Conference*, 892–896, 1998.

[7] F. Dehne. "Guest Editor's Introduction". *Algorithmica* Special Issue on "Coarse grained parallel algorithms", Vol.24, No.3/4, 173–176, 1999.

[8] L. DeVroye. Private communication. 2001.

[9] R.G. Downey, M.R. Fellows, U. Stege. "Parameterized Complexity: A Framework for Systematically Confronting Computational Intractability". *AMS-DIMACS Proceedings*, Vol.49, AMS, 49–99, 1999.

[10] R.G. Downey, M.R. Fellows. *Parameterized Complexity*. Springer-Verlag, 1998.

[11] R.G. Downey, M.R. Fellows."Fixed Parameter Tractability and Completeness I: Basic Theory". *SIAM Journal of Computing*, Vol.24, 873–921, 1995.

[12] R.G. Downey, M.R. Fellows. "Fixed Parameter Tractability and Completeness II: Completeness for $W[1]$". *Theoretical Computer Science A*, Vol.141, 109–131, 1995.

[13] R.G. Downey, M.R. Fellows. "Parameterized Computational Feasibility". *Feasible Mathematics II*, P. Clote, J. Remmel (Eds.), Birkhauser, 219–244, 1995.

[14] R.G. Downey, M.R. Fellows. "Fixed-parameter tractability and completeness." *Congressus Numerantium*, Vol.87, 161–187, 1992.

[15] M.R. Fellows. "On the complexity of vertex set problems". CS Tech. Rep., Univ. of New Mexico, 1988.

[16] M.R. Fellows. Private communication.

[17] M. Garey, D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[18] G. Gonnet. http://cbrg.inf.ethz.ch/Server/VertexCover.html.

[19] M. A. Langston. Private communication. 2002.

[20] R. Niedermeier, P. Rossmanith. "Upper Bounds for Vertex Cover Further Improved". *Proc. STACS'99*, LNCS, 1999.

[21] C.H. Papadimitriou, M. Yannakakis. "On Limited Nondeterminism and the Complexity of the V-C Dimension". *J. Comp. Syst. Sci.* 53, 161–170, 1996.

[22] C. Roth-Korostensky. "Algorithms for Building Multiple Sequence Alignments and Evolutionary Trees." PhD thesis, ETH Zrich, 2000.

[23] V.N. Rao, V. Kumar. "On the Efficiency of Parallel Backtracking". *IEEE Transactions on Parallel and Distributed Systems*, Vol.4, No.4, 427–437, 1993.

[24] U. Stege, M.R. Fellows. "An improved fixed-parameter-tractable algorithm for Vertex Cover". CS Tech. Rep. 318, ETH Zürich, 1999.

[25] U.Stege. *Resolving Conflicts from Problems in Computational Biology*. Ph.D. thesis, ETH Zürich, 2000.

[26] J.D. Thompson, D.G. Higgins, T.J. Gibson. "CLUSTAL W", *Nucleic Acids Research* 22, 4673-4680, 1994.