Solving Large FPT Problems On Coarse Grained Parallel Machines [*]

Frank Dehne [†] Andrew Rau-Chaplin [‡] Ulrike Stege [§] Peter J. Taillon [¶]

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 and application specific "plug-in" code for the k-Vertex Cover problem using C and the MPI communication library, and tested it on a network of 10 Sun SPARC workstations. This is the first experimental examination of parallel *FPT* techniques. In our experiments, we obtain excellent speedup results. Not only do we achieve a speedup of p in most cases, many cases even exhibit a super linear speedup. The latter result implies that our parallel methods, when simulated on a single processor, also yield a significant improvement over existing sequential methods.

1 Introduction

NP-complete problems abound in many important application areas ranging from computational biology to network planning. 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 [11, 12, 13, 14, 15, 16]. *FPT* algorithms have been successful in solving *NP*-complete problems for certain problem instances of practical importance which were not feasible with previous methods [11]; see Appendix A. For example, the Computational Biochemistry Research Group at ETH Zurich (http://cbrg.inf.ethz.ch), under the direction of Gaston Gonnet, has successfully used the *FPT* approach for Vertex Cover problems arising in multiple sequence alignments for computational biochemistry research [20, 25, 27]; see Appendix A. Using this new approach, the group has been able to solve larger problem than previously possible, thereby enabling Computational Biochemistry Research that was previously impossible. In this paper, we further increase the size of problems that can be solved by showing how the *FPT* approach can be parallelized. We have implemented our parallel *FPT* method

[*]Research partially supported by the Natural Sciences and Engineering Research Council of Canada and by the Pacific Institute of Mathematical Sciences (PIMS).

[†]School of Computer Science, Carleton University, Ottawa, Canada K1S 5B6, frank@dehne.net, http://www.dehne.net.

[‡]Faculty of Computer Science, Dalhousie University, Halifax, Canada B3J 2X4, arc@cs.dal.ca, http://www.cs.dal.ca/~arc.

[§]Department of Computer Science, University of Victoria, Victoria, Canada V8W 3P6, stege@csr.uvic.ca, http://www.csr.uvic.ca/~stege.

[¶]School of Computer Science, Carleton University, Ottawa, Canada K1S 5B6, ptaillon@scs.carleton.ca.

and application specific "plug-in" code for the k-VERTEX COVER problem using C and the MPI communication library, and tested it on a network of 10 Sun SPARC workstations. This is the first experimental examination of parallel *FPT* techniques. Our experiments show linear speedup or better on a range of data sets including (a) random graphs, (b) grid graphs which are very hard instances for the VERTEX COVER problem, and (c) graphs from Gaston Gonnet's Computational Biochemistry Research group. A very interesting property of our parallel *FPT* approach is that it exhibits super linear speedup for many problem instances. Our experiments show that our new parallel *FPT* approach does in fact also lead to better sequential *FPT* algorithms.

For scientists and engineers who have *NP*-complete problems to solve, the real test for any new method is how large a problem it can solve. The experimental analysis of our new parallel *FPT* approach shows that it can solve problems of a size larger than in any previously reported experiment. In [11], the authors consider the k-VERTEX COVER problem solvable for $k \leq 200$. In contrast, our parallel VERTEX COVER code, run on a standard SUN SPARC network with 10 processors, is able to solve instances of the k-VERTEX COVER problem with $k = 400$ in about 7 minutes of CPU time. This is a large improvement since the time of the sequential *FPT* algorithm for the k-VERTEX COVER problem grows exponentially in k. In order to facilitate future comparisons of different codes for solving the k-VERTEX COVER problem, we are compiling a benchmark data set that will be made available in the final version of this paper.

We now briefly review the sequential *FPT* approach. More details can be found in Appendix A. In contrast to classical complexity theory [18], parameterized complexity analysis views the input to an algorithm as consisting of two parts, (x, y), where x is the main component and y is a fixed parameter dictated by the nature of the problem at hand. The goal is to isolate, in the parameter y, the component of the input that causes the exponential time. Given a problem instance, an *FPT* algorithm is characterized by a running time $f(k) \cdot n^{\alpha}$, where $|x| = n$, $|y| = k$, α and k are constants independent of n, and where f is an arbitrary function (e.g. $f(k) = 2^k$). The two fundamental algorithmic techniques for solving *FPT* problems are *kernelization* and *bounded tree search* [12]. 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. The second phase, bounded tree search, then attempts to solve the latter problem by exhaustive search, typically requiring time exponential in k. Appendix A discusses this approach in detail for the k-VERTEX COVER problem.

In [3, 7], first definitions were formulated for efficiently parallelizable parameterized problems (*PNC* and *FPP*; see Appendix B.1). These definitions aim at parallelizing the kernelization phase of the *FPT* algorithms and leave the bounded tree search unchanged. An EREW-PRAM algorithm for the k-VERTEX COVER problem with time complexity $4 \log n + O(k^k)$, using n^2 processors, was presented in [7]. Unfortunately, the practical reality of *FPT* algorithms is rather different. Typical sequential *FPT* algorithms spend minutes on the kernelization phase and hours or days on the bounded tree search. All previous approaches [3, 7] parallelize the kernelization but do not parallelize the tree search. This is obviously not the best approach for obtaining maximum speedup through parallelization.

In this paper, we demonstrate the potential of parallelism when applied to the bounded tree search phase of *FPT* algorithms. We present a general methodology for parallelizing the bounded tree search phase of *FPT* algorithms. We also improve on the previous definition of *PNC* and *FPP*. Our experiments suggest the definition of a new, more practical, class FPT^p of *parallelizable FPT problems* which is based on the speedup obtained for the entire *FPT* algorithm (Appendix B.2). We apply our methodology to the k-VERTEX COVER problem which has important applications in multiple sequence alignments for computational biochemistry. In fact, for ease of presentation, we introduce our tree search parallelization method by describing its application to the k-VERTEX COVER problem (Section 2). The generalization to parallel tree search for other *FPT* algorithms is straight forward.

Our parallel *FPT* method is designed for the CGM (Coarse Grained Multicomputer [8, 9]) and BSP (Bulk-Synchronous Parallel [28]) machine models. A CGM simply consists of p processors, P_0, P_1, …, P_{p-1}, connected via any communication network or shared memory. Each processor has $O(N/p)$ local memory where N refers to the total problem size. Consult [8, 9, 28] for more details. Note that, the previous results [3, 7] apply to the theoretical PRAM model only.

Our parallel methods are portable and can be run efficiently on most commercially available parallel machines, including Beowulf-style clusters [2] and networks of workstations. This makes our method readily available to all computational scientists and engineers who have access to parallel hardware ranging from networks of workstations to high-end supercomputers. To illustrate this point, we are using for our own experiments only a very modest hardware platform. The observed performance, reported in this paper, will be further improved on a faster machine.

2 Coarse Grained Parallel Kernelization And Bounded Tree Search For The k-Vertex Cover Problem

Most sequential *FPT* algorithms consist of two phases, kernelization and bounded tree search [12]. The main result of this paper is an efficient parallelization of both of these phases. In this section, we describe our general methodolgy using the example of the well know k-Vertex Cover problem. For a long list of other *FPT* problems that can be solved via kernelization and bounded tree search see [12].

The Vertex Cover problem is defined as follows: 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. The k-Vertex Cover problem is a classical *FPT* problem (with fixed parameter is k) and various sequential *FPT* algorithms have been proposed. For further discussion please consult Appendix A.

We present a coarse grained parallel k-Vertex Cover algorithm which parallelizes the sequential *FPT* algorithm described in [1]. Note that, [1] combines Buss' kernelization algorithm with a 3-level, depth-first search strategy that produces a 3-ary search tree (see Appendix A). In the following two sections we describe our parallelization of the kernelization and the tree search, respectively.

2.1 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 $\langle G, k \rangle$.

In the parallel setting, this operation reduces to $O(1)$ parallel integer sorts where edges are sorted by vertex id in order to indentify the vertices with $deg(v) > k$. This sort can be implemented via deterministic sample sort [5]. Note that other kernelization rules can be applied as described in [11] and [1]. These rules are also easily reduced to $O(1)$ parallel integer sorts.

[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.

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 [5].
(1.2) Output either a kernelized graph $\langle G' = (V', E'), k'\rangle$, or VC ($\leq k$), or *"No"*.
— End of Algorithm —

Lemma 1 *Algorithm 1 performs kernelization in time $O(\frac{kn}{p})$ using $O(1)$ h-relations for communication between processors.*

2.2 Parallel Bounded Tree Search

As previously discussed, typical *FPT* implementations spend minutes on the kernelization and hours on the tree search. An efficient parallelization of the tree search is therefore of great importance.

Let VC be a set of vertices in the current vertex cover and let $\langle G'' = (V'', E''), k''\rangle$ be a problem instance associated with a node x of the search tree. In [1], the following steps are repeated until either a VC is found, or it is determined that G does not have a k-cover: (1) Randomly select a vertex, $v \in V''$. (2) Starting from v, perform a depth-first search traversing at most three edges. (3) Based on the possible paths derived from the search in Step 2, either expand node x into three children (cases 1, 2), or process immediately (cases 3, 4):
Case 1. 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:
(a) $\langle G''' = (V'' - \{v, v_2\}, E'''), k''' = k'' - 2\rangle$; $VC = VC \bigcup \{v, v_2\}$
(b) $\langle G''' = (V'' - \{v_1, v_2\}, E'''), k''' = k'' - 2\rangle$; $VC = VC \bigcup \{v_1, v_2\}$
(c) $\langle G''' = (V'' - \{v_1, v_3\}, E'''), k''' = k'' - 2\rangle$; $VC = VC \bigcup \{v_1, v_3\}$
Case 2. A 3-cycle consisting of the following sequence of vertices v, v_1, v_2, v. Associate three children with node x as follows:
(a) $\langle G''' = (V'' - \{v, v_1\}, E'''), k''' = k'' - 2\rangle$; $VC = VC \bigcup \{v, v_1\}$
(b) $\langle G''' = (V'' - \{v_1, v_2\}, E'''), k''' = k'' - 2\rangle$; $VC = VC \bigcup \{v_1, v_2\}$
(c) $\langle G''' = (V'' - \{v, v_2\}, E'''), k''' = k'' - 2\rangle$; $VC = VC \bigcup \{v, v_2\}$
Case 3. 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. 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: $\langle G''' = (V'' - \{v, v_1\}, E'''), k''' = k'' - 1\rangle$; $VC = VC \bigcup \{v\}$.

Our basic approach for parallelizing the tree search is quite simple. 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. We then assign each of the p leaves to one processor and let each processor continue searching the tree from its respective leaf. We assure that the tree search is well-randomized: that is, when a processor proceeds downwards in the search tree, it selects a random node among the still unexplored children. 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, until there are exactly p leaves $\gamma_1 \ldots \gamma_p$. Every processor, P_i, $1 \leq i \leq p$, computes the unique path in T from the root to leaf γ_i. Let (G''_i, k''_i), $1 \leq i \leq p$, be the subgraphs and updated parameters associated with γ_i.

(2.2) Processor P_i, $1 \leq i \leq p$, starts with (G_i'', k_i'') and expands/searches the subtree below γ_i in a randomized, depth-first fashion as follows:

> Processor P_i generates the children of its current problem instance as described in Cases 1-4 listed above. It then 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 the 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 $l_{par} = \min_{1 \leq i \leq p} l_i$, 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. In general, what kind of speedup can we expect?

3 Performance Analysis

3.1 Preliminary Simulation

Prior to implementation, 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 the sequential algorithms traverses the search tree. Assume, for this experiment, that there are m solutions in the search tree which are randomly distributed (with uniform distribution) over the search path Λ. For our experiment, we build an array of p rows and $n = L/p$ columns. The ith row corresponds to Λ_i and the entire array corresponds to Λ. We mark m random array elements as solutions and measure l_{seq} and $l_{par} = \min_{1 \leq i \leq p} l_i$.

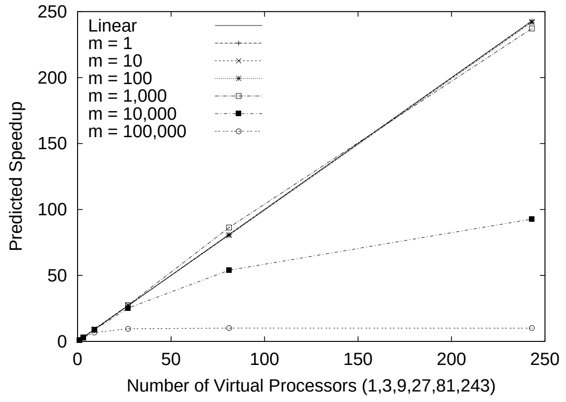

Figure 1: Simulated Speedup Estimation Through "Balls in Bins" Experiment.

The results are shown in Figure 1. The 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. 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 a 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 can be explained as follows [10]. 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 $l_{par} = \min_{1 \le i \le p} l_i$ that need to be traversed by the parallel algorithm is bounded by $E(l_{par}) \le \frac{L/p}{m+1} + p$. Therefore, we obtain for the speedup

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

For $m \ll L/p$ the second part of the denominator becomes neglectable and we get an expected speedup $E(s_p)$ of approximately p. This is what we observed in Figure 1 for $m \le 1000$. 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 [19] 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.

3.2 Performance Analysis Of Our Parallel Implementation

We have implemented our parallel *FPT* method and application specific "plug-in" code for the k-VERTEX COVER problem using C and the MPI communication library. The code is about 1200 lines in length and was developed by a single programmer in about 3 months. Our code was tested on a cluster of 10 Sun Sparc-10 workstations. Each machine had a 440MHz Ultra Sparc II processor, 256MB of RAM, 2MB CPU cache, and 8GB hard disk space. The machines were interconnected with 100MBps Ethernet, through a 12 port 10/100MBps hub. The operating system was Sparc Solaris 7 and we used LAM/MPI-6.3.2 as our MPI platform. Note that, we are using a very modest hardware platform for our experiments and our observed performance will be further improved on a faster machine. The workstation cluster used actually forms the backbone of the graduate student computing facilities, and our experiments were also subject to background load. The timing was measured using the Unix system call *times*, which returns the accumulated CPU time of the user-process.

Experiments were run on three different sets of test data: random graphs, grid graphs, and Gonnet's graphs. For each graph instance, 20 experiments each were run for $p = 1$, 3, 9, 27, 81, 243. Note that, p represents the number of virtual processors created by the

MPI system. The number of physical processors was always equal to 10. The values for p are all powers of 3 since the bounded search tree has degree 3. As we will soon observe, our method achieves super linear speedup for many problem instances. In such cases, it is beneficial to have many more virtual processors than physical ones. In fact, super linear speedup implies that our method is an improvement of the sequential method.

Random Graphs

We implemented a graph generator for random graphs which creates graphs of various sizes and edge densities. The speedup obtained by our parallel k-Vertex Cover code are shown in Figure 2. Each figure shows the results for five random graphs of the same size and number of edges, using $p = 1, 3, 9, 27, 81, 243$ virtual processors. The straight line labeled "linear" represents linear speedup. Most strikingly, observe the many cases of super linear speedup. This effect is caused by a non-uniform distribution of k-Vertex Cover solutions in the search tree and consistent with the discussion in Section 3.1. We also observe, in some cases, a decreasing speedup for $p > 81$ This decrease in speedup is caused by an increasing overhead of the MPI platform to manage the larger number of virtual processors and the fact that m is getting closer to L/p as p increases.

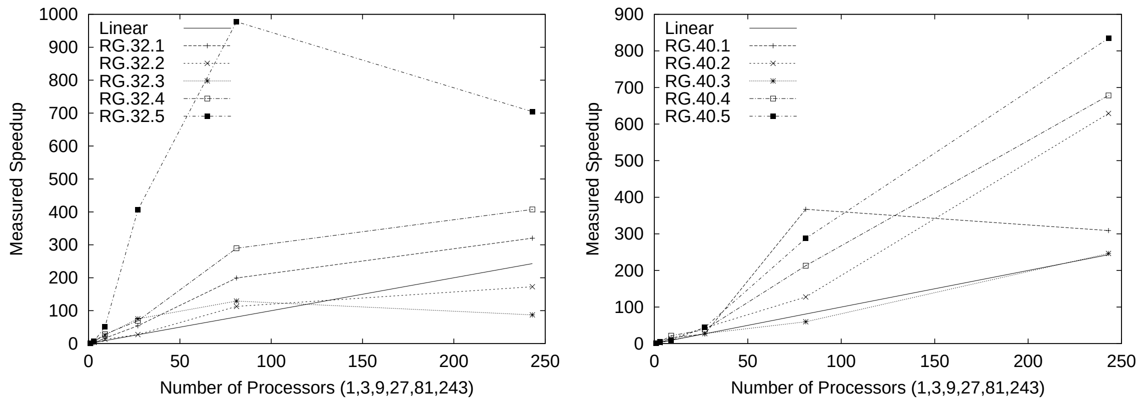

Figure 2: Average speedup measured for (a) random graphs RG.32.1, …, RG.32.5 ($|V| = 700$, $|E| = 1000$, $|VC| = 32$) and (b) random graphs RG.40.1, …, RG.40.5 ($|V| = 700$, $|E| = 1000$, $|VC| = 40$). Each curve represents experiments on the same random graph for different numbers of processors. Each data point represents the average of 20 experiments on the same graph.

Grid Graphs

We implemented a graph generator for grid graphs which consist of a lattice of vertices, each of them connected to their four adjacent neighbors. Such graphs represent particularly hard cases for the k-Vertex Cover problem because the kernelization phase yields no reduction in graph size whatsoever. Note that, for a grid graph the size of the minimum vertex cover is exactly $|V|/2$.

The speedup obtained by our parallel k-Vertex Cover code are shown in Figure 3. Each figure shows the results for the grid graph of a given size, using $p = 1, 3, 9, 27, 81, 243$ virtual processors. The straight line labeled "linear" represents linear speedup. We observe super linear speedup for all cases measured. The growth in speedup appears to increase initially (up to $p = 81$) and then the curves become flatter for larger number of virtual processors, p. This effect is caused by a tradeoff between (1) the overhead incurred in using large numbers of virtual processors as well as the fact that m is getting closer to

L/p as p increases and (2) the benefit provided by more virtual processors due to super linear speedup.

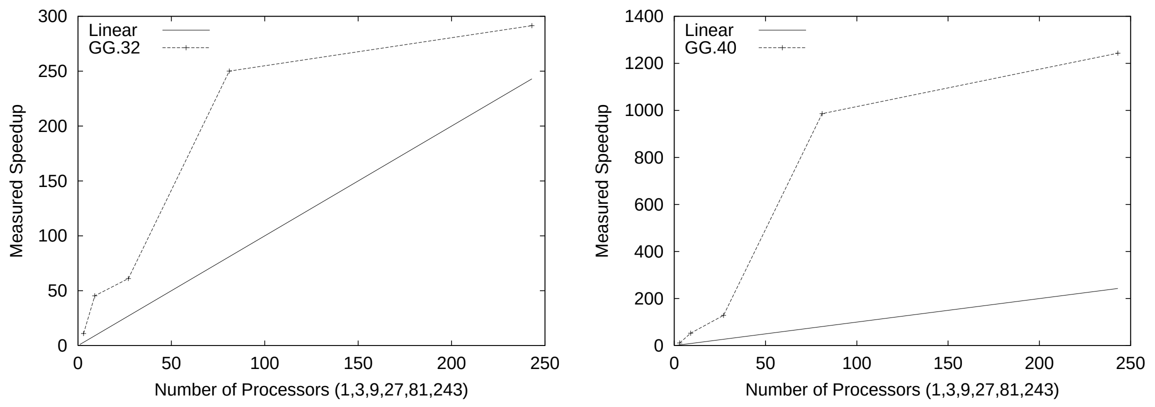

Figure 3: Average speedup measured for (a) grid graph GG.32 ($|V| = 64$, $|E| = 112$, $|VC| = 32$) and (b) grid graph GG.40 ($|V| = 81$, $|E| = 144$, $|VC| = 40$). Each data point represents the average of 100 experiments.

Gonnet's Graphs

The Computational Biochemistry Research Group at ETH Zurich (http://cbrg.inf.ethz.ch), under the direction of Dr. Gaston Gonnet, has implemented a Vertex Cover algorithm for computational biology research which combines the sequential *FPT* approach with a heuristic method [20]. It is important to note that Gonnet's method does not guarantee a correct result as it is a heuristic. A main feature of (sequential and parallel) *FPT* methods is that they are guaranteed to report the correct result. Gonnet's graphs behave similar to grid graphs in that the kernelization phase yields only a very small reduction in the size of these graphs. The speedup obtained by our parallel k-Vertex Cover code are shown in Figure 4. The four curves shows the results for graphs G.203, G.205, G.293, G.299, G.300, and G.304 from http://cbrg.inf.ethz.ch. Each figure shows the measured speedups for $p =$ 1, 3, 9, 27, 81, 243 virtual processors. The straight lines labeled "linear" represents linear speedup. Again, we observe super linear speedup for all cases measured. The speedup appears to grow very quickly and, in some cases, reaches one or two orders of magnitude above p.

References

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

[2] D.J. Becker, T. Sterling, D. Savarese, J.E. Dorband, U.A. Ranawak, C.V. Packer. "Beowulf: A Parallel Workstation for Scientific Computation". In *Proceedings of the International Conference on Parallel Processing*, 1995.

[3] H.L. Bodlaender, R.G. Downey, M.R. Fellows. "Applications of Parameterized Complexity to Problems of Parallel and Distributed Computation". Unpublished extended abstract, 1994.

[4] J.F. Buss, J. Goldsmith. "Nondeterminism within *P*". In *SIAM Journal of Computing*, Vol.22, pp. 560–572, 1993.

[5] A. Chan, F. Dehne, "A Note on Coarse Grained Parallel Integer Sorting". In *Proc. 13th Annual Int. Symp. on High Performance Computers (HPCS'99)*, Kingston, Canada, 1999, pp. 261–267.

[6] J. Chen, I.A. Kanj, W. Jia. “Vertex Cover: Further observations and further improvements”. In *25th International Workshop on Graph-Theoretical Concepts in Computer Science (WG’99)*, LNCS, 1999.

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

[8] F. Dehne, “Guest Editor’s Introduction”. In *Algorithmica* Special Issue on “Coarse grained parallel algorithms”, Vol.24, No.3/4, July/August 1999, pp. 173–176.

[9] F. Dehne, A. Fabri, A. Rau-Chaplin, “Scalable parallel computational geometry for coarse grained multicomputers,” International Journal on Computational Geometry, Vol. 6, No. 3, 1996, pp. 379 - 400.

[10] L. Devroye, private communication, GRACO Conference, Fortaleza, Brasil, March 2001.

[11] R.G. Downey, M.R. Fellows, U. Stege. “Parameterized Complexity: A Framework for Systematically Confronting Computational Intractability”. In *Contemporary Trends in Discrete Mathematics: From DIMACS and DIMATIA to the Future*, F. Roberts, J. Kratochvil, and J. Nesetril, eds., *AMS-DIMACS Proceedings Series*, Vol.49, AMS, pp. 49–99, 1999.

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

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

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

[15] R.G. Downey, M.R. Fellows. “Parameterized Computational Feasibility”. In *Feasible Mathematics II, P. Clote, J. Remmel (eds.)*, Birkhauser, Boston, pp. 219–244, 1995.

[16] R.G. Downey, M.R. Fellows. “Fixed-parameter tractability and completeness.” In *Congressus Numerantium*, Vol.87, pp. 161–187, 1992.

[17] M.R. Fellows. “On the complexity of vertex set problems”. Technical report, Computer Science Department, University of New Mexico, 1988.

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

[19] V. Nageshwara Rao, Vipin Kumar. “On the Efficiency of Parallel Backtracking”. IEEE Transactions on Parallel and Distributed Systems, Vol. 4, No. 4, 1993, pp. 427–437.

[20] G. Gonnet. “Vertex cover heuristic”, http://cbrg.inf.ethz.ch/VertexCover.html.

[21] J. Jaja. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[22] R. Niedermeier, P. Rossmanith. “Upper Bounds for Vertex Cover Further Improved”. In *Proceedings of the 16th Symposium on Theoretical Aspects in Computer Science (STACS’99)*, LNCS, 1999.

[23] R. Niedermeier, P. Rossmanith. “A General Method to Speed Up Fixed-Parameter-Tractable Algorithms”. Technical Report TUM-I9913, Institut für Informatik, Technische Universität München, 1999.

[24] C.H. Papadimitriou, M. Yannakakis. “On Limited Nondeterminism and the Complexity of the V-C Dimension”. In *Journal of Computer and System Sciences*, Vol.53, pp. 161–170, 1996.

[25] C. Roth-Korostensky. “Algorithms for Building Multiple Sequence Alignments and Evolutionary Trees.” Phd thesis, ETH Zrich, Institute of Scientific Computing, February 2000.

[26] U. Stege, M.R. Fellows. “An improved fixed-parameter-tractable algorithm for Vertex Cover”. Technical Report 318, Department of Computer Science, ETH Zürich, April 1999.

[27] U.Stege. *Resolving Conflicts from Problems in Computational Biology*. Ph.D. thesis, No. 13364, ETH Zürich, 2000.

[28] L. Valiant. “A bridging model for parallel computation”. *Communication of the ACM*, Vol.33, No.8, August, 1990.

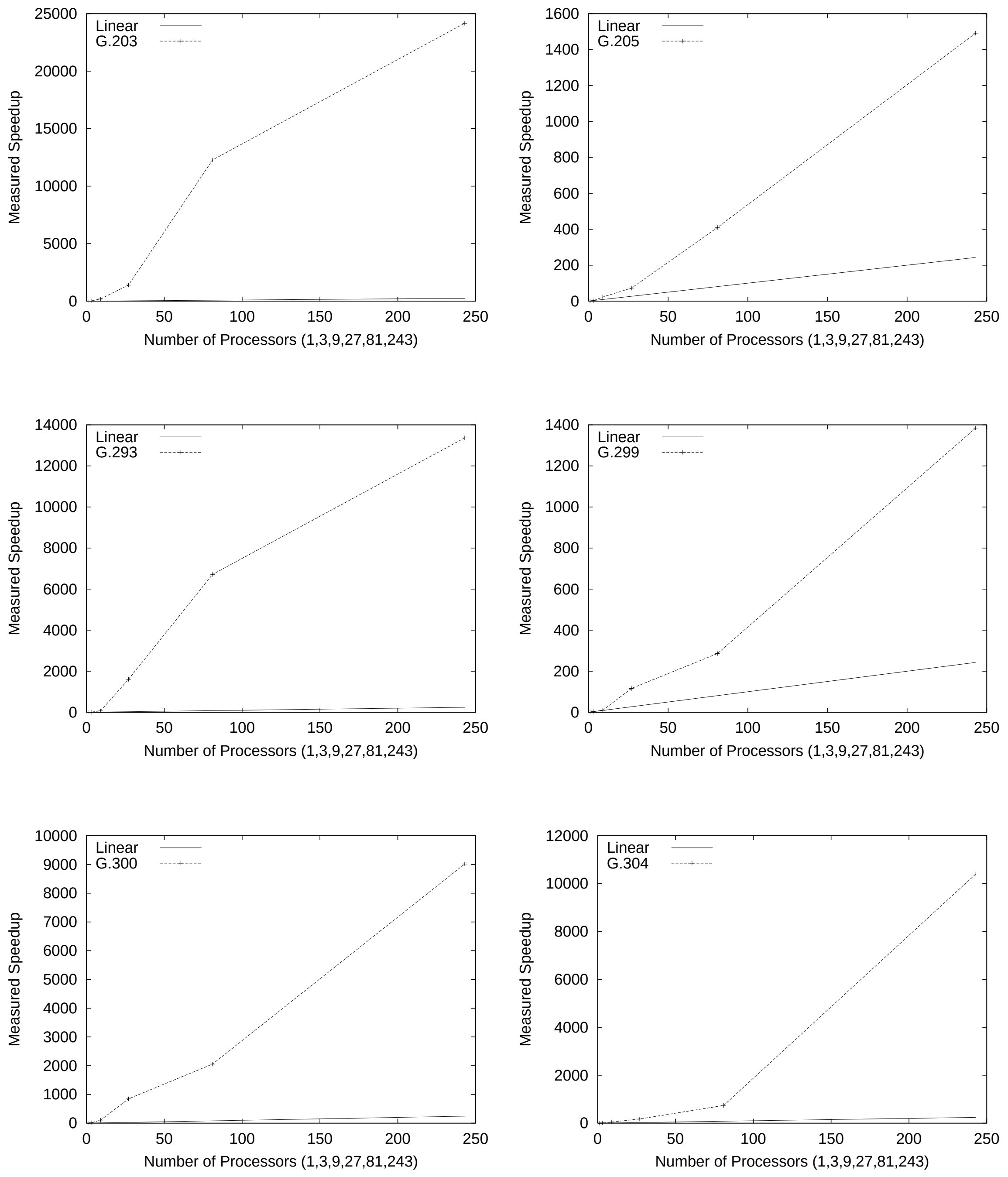

Figure 4: Average speedup measured for Gonnet's graphs (http://cbrg.inf.ethz.ch): (a) G.203 ($|V| = 60$, $|E| = 246$, $|VC| = 41$), (b) G.205 ($|V| = 60$, $|E| = 246$, $|VC| = 41$), (c) G.293 ($|V| = 62$, $|E| = 256$, $|VC| = 43$), (d) G.299 ($|V| = 65$, $|E| = 272$, $|VC| = 43$), (e) G.300 ($|V| = 65$, $|E| = 272$, $|VC| = 45$). (f) G.304 ($|V| = 65$, $|E| = 272$, $|VC| = 45$). Each data point represents the average of 20 experiments on the same graph.

Appendix

A Fixed Parameter Tractability (FPT) And The k-Vertex Cover Problem

Fixed-parameter tractability (FPT) has been proposed in [11, 12, 13, 14, 15, 16] 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 $f(k) \cdot n^\alpha$, where $|x| = n$, $|y| = k$ is a parameter, α is a constant independent of n and k, and f is an arbitrary function. In many cases, FPT algorithms use kernelization and bounded tree search, usually resulting in a running time of $f(k) + n^\alpha$. It was shown in [11] that a problem is in FPT if and only if it is kernelizable.

Although nearly half the NP-Complete problems in [18] have been shown to be FPT [11], not all problems admit a parametric solution. For example, the best algorithm to solve the Dominating Set problem is exponential in n and k. For parameterized complexity, the analog of NP-hardness is hardness for $W[1]$; see [14]. Dominating Set is hard for $W[1]$ and is therefore unlikely to be fixed-parameter tractable.

The Vertex Cover problem is defined as follows: 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.

The k-Vertex Cover problem has important applications in multiple sequence alignments for computational biochemistry [27]. 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 which is 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 the Vertex Cover of the conflict graph.

The Vertex Cover problem is known to be NP-Complete [18], but in the context of parameterized complexity [11, 12, 13, 14, 15] the problem is fixed-parameter tractable. Consider the following k-Vertex Cover kernelization algorithm by Buss [4]: given a graph $G = (V, E)$ and a parameter k, find the set S consisting of all vertices v such that $deg(v) > k$. Let $|S| = b$. If $b > k$ then we conclude 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 (and all their incident edges from E). 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, the graph G', which is called kernelized, has a vertex set V' bounded in size by $O(k^2)$.

The next phase, bounded tree search [12], is based on an exhaustive combinatorial search. The search tree is a rooted tree and bounded in size by a function $f(k)$. The nodes of the search tree are labeled by k-solution candidate sets. Consider the following k-Vertex Cover algorithm by Fellows [16, 17]: observe that, given a graph $G = (V, E)$, for each $v \in V$ and each vertex cover VC of G, either $v \in VC$ or $N(v) \subseteq VC$ [2]. Thus, given an instance $\langle G, k \rangle$ for the k-Vertex Cover problem, the original input graph G has a k-vertex cover if $\langle G - v, k - 1 \rangle$ or $\langle G - N(v), k - |N(v)| \rangle$ has a solution. Since the parameter k reduces in each such step by at least one, we can decide in time $O(2^k |V|)$ whether G has a vertex cover of size k.

The first Vertex Cover algorithm is due to Buss and has an $O(kn + 2^k k^{2k+2})$ time complexity [4]. Papadimitriou and Yannakakis, while proving that k-Vertex Cover $\in P$ when k is restricted to $O(\log n)$, provided an $O(3^k n)$ algorithm using maximal matchings [24]. Downey and Fellows presented a different algorithm that runs in time $O(kn + 2^k k^2)$ [14]. Balasubramanian, Fellows, and Raman suggested two different FPT algorithms in

[2] $N(v)$ = the set of vertices that constitute the neighborhood of vertex v. $N[v] = N(v) \bigcup \{v\}$.

their publication [1]. The running times of the algorithms are $O((\sqrt{3})^k k^2 + kn)$ and $O((1.324718)^k k^2 + kn)$, respectively. The first of these two algorithms will form the basis of the parallel algorithm described in this paper. The second algorithm has been subsequently improved by Downey, Fellows, and Stege by using a better kernelization of the input graph to obtain a running time of $O(kn + r^k k^2)$, $r \approx 1.3195$ [11]. An algorithm by Niedermeier and Rossmanith runs in time $O(kn + r^k k^2)$, $r \approx 1.2917$, using an improved search tree [22]. Recently, Stege combined the results of [11], [22] and, using an improved kernelization and search tree, developed an algorithm with running time of $O(kn + r^k k)$, $r \approx 1.2906$ [26, 27]. A further improvement was made in [6], where the running time was reduced to $O(kn + 1.271^k k^2)$.

B Parallel Parameterized Complexity Classes

B.1 Previous Proposals: *PNC*, *FPP*

The notion of parallel fixed-parameter tractability was first introduced in [3]. Cesati and Di Ianni expanded on this preliminary discussion to introduce the parallel fixed-parameter tractable complexity classes *PNC* and *FPP* [7]. They also propose the first *FPT* EREW-PRAM algorithm for solving the k-Vertex Cover problem in time $4 \log n + O(k^k)$, using n^2 processors.

Recall that, the class *NC* constitutes the set of problems for which there exist an efficient PRAM algorithm. More formally, the class NC^k, $k > 1$, is the class of all problems solvable in $O(\log^k n)$ time, using $n^{O(1)}$ processors, where n is the length of the input, and k is a constant independent of n [21]. Let $\langle x, k \rangle$ be a problem instance, where k is the parameter, f, g and h are arbitrary functions, and α and β are constants independent of x and k. Bodlaender, et al. [3] define the parameterized analog of *NC*, called *PNC*, as the class of parameterized problems solvable by a parallel algorithm in time $f(k)(\log |x|)^{h(k)}$, using at most $g(k)|x|^\beta$ processors. Since the exponent of the logarithmic term is a function of k and can grow very quickly, Cesati and Di Ianni [7] proposed an alternate definition of fixed-parameter parallelizable problems. They define *FPP* as the class of parameterized problems solvable by a parallel algorithm in time $f(k)(\log |x|)^\alpha$, using at most $g(k)|x|^\beta$ processors.

B.2 New Proposed Parallelizable *FPT*: FPT^p

The definitions of *FPP* and *PNC* in [3, 7] imply that $FPP \subseteq PNC \subseteq FPT$ which makes them nicely consistent with $NC \subseteq P$. Unfortunately, the definitions of *FPP* and *PNC* vis-a-vis *FPT* do not capture the notion of satisfactory parallelization in the same way as the definition of *NC* vis-a-vis *P*. For example, the EREW-PRAM k-Vertex Cover algorithm presented in [7] has running time $4 \log n + O(k^k)$, using n^2 processors, which implies that k-Vertex Cover $\in$ *FPP*. However, the running time of the parallel algorithm is no improvement over the sequential algorithm. Sequential *FPT* algorithms for k-Vertex Cover , when implemented, spend minutes on the kernelization phase and hours or days on the bounded tree search. The approach in [3, 7] parallelizes the kernelization but does not parallelize the tree search. The speedup obtained is negligible.

Another shortcoming of the *FPP* and *PNC* definitions in [3, 7] is that they are for the PRAM model only. It is well known that many PRAM algorithms, when implemented on an actual parallel machine perform very poorly. The parallel processing community has developed much more realistic models like the BSP [28], and CGM [8, 9], which yield much better performance in practice.

We now define a new class FPT^p_α of *parallelizable FPT* problems. Consider a parallel machine model α (e.g., α = CGM) and a problem $L \in FPT$. The problem L is in FPT^p_α

if there exists a parallel algorithm that solves L in expected time T_p for p processors such that $T_1 = f(k) \cdot n^\beta$ and $T_p = O(\frac{T_1}{p})$.

The above definition of FPT^p_α is straight-forward. It simply asks that the parallel FPT algorithm be p times as fast as the respective sequential FPT algorithm. It thereby addresses the shortcoming of FPP and PNC discussed above, that FPP and PNC algorithms can have negligible parallel speedup. Our definition of FPT^p_α also adds the dimension of the parallel machine model which is of paramount importance in parallel computing. Note that, $FPT^p_{CGM} \subset FPT^p_{BSP} \subset FPT^p_{PRAM}$. We define $FPT^p = FPT^p_{CGM}$ as the class of *parallelizable* FPT problems which have an expected speedup of p for all of the above models.