ACADEMIC PRESS

Available at www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

Journal of Computer and System Sciences 67 (2003) 691–706

JOURNAL OF COMPUTER AND SYSTEM SCIENCES

http://www.elsevier.com/locate/jcss

Solving large FPT problems on coarse-grained parallel machines ☆

James Cheetham,[a] Frank Dehne,[b,*] Andrew Rau-Chaplin,[c] Ulrike Stege,[d] and Peter J. Taillon[b]

[a] *Institute of Biochemistry, Carleton University, Ottawa, Canada K1S 5B6*
[b] *School of Computer Science, Carleton University, Ottawa, Canada K1S 5B6*
[c] *Faculty of Computer Science, Dalhousie University, Halifax, Canada B3J 2X4*
[d] *Department of Computer Science, University of Victoria, Victoria, Canada V8W 3P6*

Received 30 September 2001; revised 16 April 2002

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 in, for example, the analysis of 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 32-node Beowulf cluster. This is the first experimental examination of parallel FPT techniques. 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 $k \geqslant 400$ in less than 1.5 h.
© 2003 Elsevier Inc. All rights reserved.

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

* Corresponding author.

E-mail addresses: jcheetha@ccs.carleton.ca (J. Cheetham), frank@dehne.net (F. Dehne), arc@cs.dal.ca (A. Rau-Chaplin), stege@csr.uvic.ca (U. Stege), ptaillon@scs.carleton.ca (P.J. Taillon).

URLs: http://www.carleton.ca/~jcheetha, http://www.dehne.net, http://www.cs.dal.ca/~arc, http://www.csr.uvic.ca/~stege.

0022-0000/03/$ - see front matter © 2003 Elsevier Inc. All rights reserved.
doi:10.1016/S0022-0000(03)00075-8

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 "real world" problem instances at hand. Fixed-parameter tractability (FPT) is a new technique for confronting the obstacle of NP-completeness [11–16]. FPT algorithms have been successful in solving NP-complete problem instances of practical importance which were too large to be solved with previous methods [11]. 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 Zuerich has successfully incorporated the FPT approach for VERTEX COVER problems arising in multiple sequence alignments for computational biochemistry research [19,23,25]. 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. 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 32-node Beowulf cluster [2]. This is the first experimental examination of parallel FPT techniques. When tested on sequence data obtained from the National Center for Biotechnology Information (NCBI), our parallel FPT method showed good relative speedup. To explore general graph classes and known hard problem instances, we also tested our parallel FPT method on random graphs and grid graphs, respectively, where we observed good relative speedup as well.

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 it can solve. In [16], the authors consider the k-VERTEX COVER problem solvable for $k \leqslant 200$. Our parallel code is able to solve much larger "real world" instances of the k-VERTEX COVER problem. For example, we extracted 730 sequences of the src-homology domain 2 (SH2) from the NCBI database and computed the input graph for the k-VERTEX COVER problem using ClustalW [26], which has a minimum vertex cover of $k = 461$. Our parallel FPT method, executed on 27 processors of our Beowulf cluster, found this minimum vertex cover in 72:45 min. Note that, the time of the sequential FPT algorithm for the k-VERTEX COVER problem grows exponentially in k. Therefore, the increase of the solvable problem size from $k \leqslant 200$ to $k \geqslant 400$ is significant.

This paper presents 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. Our parallel FPT method is designed for the coarse-grained multicomputer (CGM) [8,9] and bulk-synchronous parallel (BSP) [27] machine models. A CGM simply consists of p processors, $P_0, P_1, \ldots, P_{p-1}$, 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 [8,9,27] for more details.

Compared to previous results on parallel FPT algorithms [3,5], which apply to the theoretical PRAM model only, our methods are portable and can be run efficiently on most commercially available parallel machines, including shared memory machines, CC-NUMA, Beowulf clusters and networks of workstations. Furthermore, the methods in [3,5] parallelize only the kernelization

phase and leave the tree search unchanged. However, typical FPT implementations often spend minutes on the kernelization and hours or days on the tree search. Hence, it is important to parallelize *both* phases. The main contribution of this paper is to provide an efficient implementation of parallel bounded-tree search.

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

2. Review: fixed-parameter tractability and the k-VERTEX COVER problem

Fixed-parameter tractability (FPT) has been proposed in [11–16] 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 $f(k) + n^{\alpha}$, 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* [15]. 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 [16] 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, typically requiring time exponential in k.

Although nearly half the NP-complete problems in [18] have been shown to be FPT [16], 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 [13]. DOMINATING SET is hard for $W[1]$ and is therefore unlikely to be fixed-parameter tractable.

The k-VERTEX COVER problem has important applications in multiple sequence alignments for computational biochemistry [25]. The VERTEX COVER problem is defined as follows [18]: 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 score below a given threshold. 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. Consult Section 4.2 for more details.

The VERTEX COVER problem is known to be NP-complete, but in the context of parameterized complexity the problem is fixed-parameter tractable [11–15]. Consider the following k-VERTEX

COVER kernelization algorithm by Buss and Goldsmith [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 that 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 [15], 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 [1,17]: 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 $N(v) \subseteq VC$.[1] 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]. Improvements have been presented in [1,13,20,21,24,25]. Recent results in [7,11] present solutions with times complexity $O(kn + r^k k^2)$ for $r \approx 1.3$. These algorithms exhibit trade-offs between small differences in r and leading constants.

3. 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 [15]. The main result of this paper is an efficient parallelization of both of these phases. In this section, we describe our general methodology using the example of the well-known k-VERTEX COVER problem. For a list of other FPT problems that can be solved via kernelization and bounded-tree search see [15].

We present a coarse-grained parallel k-VERTEX COVER algorithm which parallelizes aspects of the two sequential FPT algorithms described in [1]. The first algorithm in [1] combines Buss' kernelization technique with a three-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 [1]).

We now present a brief overview of our parallel k-VERTEX COVER algorithm, with details to follow in Sections 3.1 and 3.2.

All processors, $P_i, 0 \leqslant i \leqslant p-1$, together perform parallel kernelization on the problem instance $\langle G = (V, E), k \rangle$, and the resulting instance $\langle G', k' \rangle$ is then broadcast to all processors. Let VC_{kern} be the set of vertices determined by the kernelization phase to be in the vertex cover set, VC. Each processor, $P_i, 0 \leqslant i \leqslant p-1$, 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

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

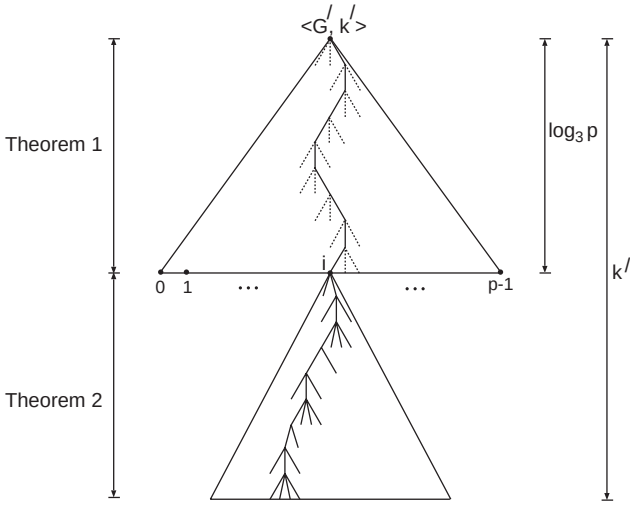

Fig. 1. Search path for processor P_i in Algorithm 2, using *Theorem* 1 and *Theorem* 2 of [1].

search-frontier splitting, as each processor now has a unique problem instance, $\langle G_i'', k_i'' \rangle$. Each processor, $P_i, 0 \leqslant i \leqslant p-1$, then locally performs a fully random depth-first search of the subtree rooted at leaf i, starting with instance $\langle G_i'', k_i'' \rangle$ (see Fig. 1). When a processor finds a solution, it outputs the set $VC_{\text{kern}} \bigcup VC_i$ 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.[2] Let $k' = k - b$. If the resulting graph, G', has more than $k \cdot k'$ edges, then we can conclude that 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 identify the vertices with $deg(v) > k$. This sort can be implemented via deterministic sample sort [6]. Note that other kernelization rules can be applied as described in [1,16]. These rules are also easily reduced to $O(1)$ parallel integer sorts.

Algorithm 1. Parallel kernelization
Input: $\langle G = (V, E), k \rangle$.
Output: $\langle G', k' \rangle$ or "No".

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

(1.1) Simulate Buss' kernelization algorithm on $G = (V, E)$ via $O(1)$ parallel integer sorts, using deterministic integer sample sort [6].
(1.2) Output either a kernelized graph $\langle G' = (V', E'), k' \rangle$, or VC($\leqslant k$), 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.*

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:

(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. 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:

(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. 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 running time 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 running time of the algorithm is $O((1.324718)^k k^2 + kn)$.

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. 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 well-randomized: that is, when a processor proceeds downwards in the search tree, it selects a random node among the still unexplored children. See Fig. 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($\leqslant k$), 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, $0 \leqslant i \leqslant p-1$, computes the unique path in T from the root to leaf γ_i. Let (G_i'', k_i''), $0 \leqslant i \leqslant p-1$, be the subgraphs and updated parameters associated with γ_i.

(2.2) Processor $P_i, 0 \leqslant i \leqslant p-1$, 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 $l_{\text{par}} = \min_{0 \leqslant i \leqslant p-1} l_i$, the minimum number of nodes that a process has to traverse until it reaches a solution node. The possible relative speedup observed corresponds to the ratio between l_{seq} and l_{par}. What relative 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 relative speedup can we expect?

A "balls-in-bins" model can be used to predict the relative 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, and 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. We mark m random array elements as solutions and measure l_{seq} and $l_{\text{par}} = \min_{0 \leqslant i \leqslant p-1} l_i$.

The expected number of nodes in Λ that need to be traversed by the sequential algorithm is given by $E(l_{\text{seq}}) = \frac{L}{m+1}$. The expected number of nodes $l_{\text{par}} = \min_{1 \leqslant i \leqslant p} l_i$ that need to be traversed by the parallel algorithm is bounded by $E(l_{\text{par}}) \leqslant \frac{L/p}{m+1} + p$ [10]. Therefore, we obtain an expected relative speedup

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

The above is only a lower bound on the expected relative speedup $E(s_p)$. We have simulated the "balls-in-bins" experiment in order to obtain a better understanding of the exact value of $E(s_p)$. The simulation results are shown in Fig. 2. The experiments were performed for $L = 1\,000\,000$, $m = 1, 10, 100, 1000, 10\,000, 100\,000$ and $p = 3, 9, 27, 81, 243$ processors. The x-axis represents the number p of processors and the y-axis represents the relative speedup $s_p = l_{\text{seq}}/l_{\text{par}}$. Each data point shown corresponds to the average of 150 experiments. The diagonal line, $s_p = p$ represents linear relative 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 interesting cases in practice because the number of actual k-Vertex Cover solutions is typically small compared to the very large, exponential size, search space. Even for $m = 10\,000$, that is where 1% of the entire search space correspond to solutions, we observe a relative speedup of about $p/2$. Only for $m = 100\,000$, that is where 10% of the entire search space correspond to solutions, we observe very low relative speedup. Note that in this case, any sequential method would find a solution in

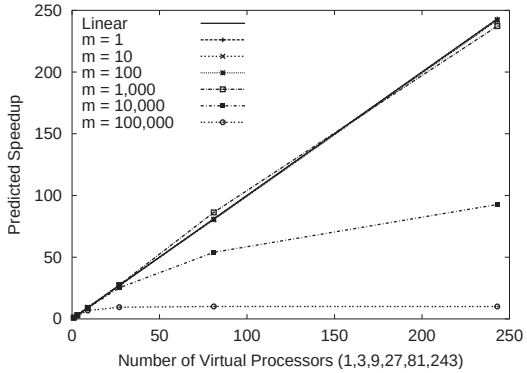

Fig. 2. Simulated relative speedup estimation through "balls-in-bins" experiment.

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 relative speedup for low density m/L observed in Fig. 2 is consistent with the bound on $E(s_p)$ derived above. For $m \ll L/p$, the second part of the denominator becomes negligible and we get an expected relative speedup $E(s_p)$ of approximately p. It is important to note that the above lower bound on $E(s_p)$ is only a coarse lower bound. The actual relative speedup can be considerably better. Furthermore, as the discussion in [22] 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 relative speedup observed is often better in the non-uniform case than in the uniform case.

4. Experimental results

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 [7,20], exhibit trade-offs 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.

Our experimental platform consisted of a 32-node Beowulf cluster with 1.8 GHz Intel Xeon processors, 512 MB RAM per node and 60 GB of disk storage per node. All nodes were interconnected via a Cisco gigabit ethernet switch. Every node was running Linux Redhat 7.2 with gcc 2.95.3 and MPI/LAM 6.5.6.

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 Beowulf cluster.

Our experiments proceeded in the following steps:

1. *Sequential Experiments* (Section 4.3)

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

2. *Parallel Experiments* (Section 4.4)

(a) `Code-p` *Parallel wall clock times*: We executed `Code-p` on 27 processors of our parallel machine and measured the parallel wall clock time.
(b) `Code-p` *Relative speedup*: We executed `Code-p` on 1, 3, 9, and 27 processors of our parallel machine 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.

4.2. Data sets

For our experiments, we primarily relied on test data from the NCBI (http://www.ncbi.nlm.nih.gov/). We obtained and processed various sets of amino acid sequences. For biologists, sequence alignments are a very useful computational tool because alignments can be used to infer evolutionary relationships among genes and proteins. 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.

A typical experiment involves a large set of amino acid sequences for an orthologous protein from distantly related organisms which are suspected to have a common ancestor. The task is to remove a minimum number of sequences (organisms) from the set that contradict the common ancestor hypothesis. Alignments can also be used to determine the order in which variations in sequences occurred, to infer when gene duplication events occurred, and to identify amino acid residues necessary for protein functions.

To test our algorithm, sets of amino acid sequences were collected from the NCBI database. Several protein modules that comprise large families of sequences (organisms) were chosen for alignments. The data sets selected are listed in Table 1: 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. Protein kinases comprise a large and important family of enzymes involved in cellular regulation. SH2 domain protein modules are involved in targeting proteins to specific sites in cells by binding to phosphortyrosine. Thrombin is a protease involved in the blood coagulation cascade and promotes blood clotting by converting fibrinogen to fibrin. pleckstrin homology domain (PHD) is a protein domain about 100 amino acid residues in length that 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

Table 1
Sequences used and resulting graph sizes

Data set	Threshold	$\|V\|$	$\|E\|$	$k = \|VC\|$	k'
Somatostatin	10	559	33 652	273	255
WW	10	425	40 182	322	318
Kinase	16	647	113 122	497	397
SH2 (src-homology domain 2)	10	730	95 463	461	397
Thrombin	15	646	62 731	413	413
PHD (pleckstrin homology domain)	10	670	147 054	603	603

Table 2
Random and grid graphs used

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

ranks the scores of the pairwise alignments. The conflict graph, i.e. the input for the minimum 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. The thresholds values used are shown in Table 1, together with the sizes of the resulting conflict graphs and the values of k and k'.

To also 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.

4.3. Performance results: sequential experiments

We executed `Code-s` on a single processor of our parallel machine and measured the sequential wall clock time (see Fig. 3, first set of vertical bars). Each data point in Fig. 3 represents the average of five experiments. We selected data sets Somatostatin, WW, PHD, random and grid (Tables 1 and 2) because they would complete on a single processor in a reasonable amount of time. We observe that the wall clock times for `Code-s` shown in Fig. 3 do not strictly increase with either k or k'. The structure of the graphs is clearly an important factor for the performance of `Code-s`.

While developing `Code-p`, we used simulation mode on a single processor for development/debugging purposes. In this mode, MPI/LAM simulates p virtual processors as independent processes on the same physical processor. We observed that `Code-p` simulated on a single processor would, for small numbers of virtual processors, often run faster than `Code-s`. It appeared that the simulated parallel code, exploring the search tree from multiple starting points, would often find a solution quicker than the sequential code. Note that, in simulation mode, the sequential wall clock time for `Code-p` is the sum of the wall clock times of the individual processes

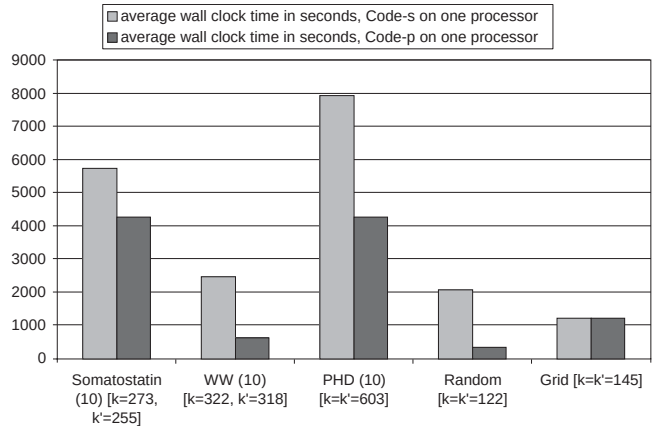

Fig. 3. Average sequential wall clock times.

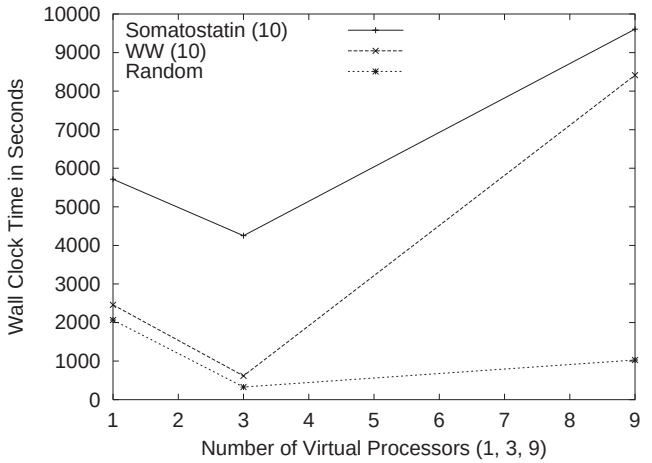

Fig. 4. `Code-p` on one processor. Average sequential wall clock time as a function of the number of simulated virtual Processors.

plus the overhead created e.g. by the context switches and MPI/LAM. As illustrated in Fig. 4, there is typically a net benefit only for a small number of virtual processors. Each data point in Fig. 4 represents the average of five experiments. For most of our data sets, the minimum average wall clock time is achieved by using three virtual processors. We conclude that, exploring the search tree from multiple starting points as proposed in our parallel FPT method can also lead to improvements of sequential FPT methods. Fig. 3 compares the average wall clock time for `Code-s` on one processor with the minimum average wall clock time for `Code-p` on one processor in simulation mode. In each case, except for Grid, we observe that `Code-p` simulated on one processor runs faster than `Code-s` on one processor for the same data set. In some cases, the difference is substantial. For the relative speedup measurements in the following Section 4.4, the "baseline" (i.e. time for one processor) was therefore set to the minimum of the wall clock times measured for `Code-s` and one processor simulations of `Code-p`.

4.4. Performance results: parallel experiments

We executed `Code-p` on 27 processors of our parallel machine and measured the average parallel wall clock time. Fig. 5 shows the results for the Somatostatin, WW, Kinase, SH2, Thrombin and PHD data sets. Each data point represents the average of ten experiments. We observe that our parallel FPT method is able to solve "real world" problem instances of size $k \geqslant 400$ in less than 1.5 h, whereas previously, for sequential FPT methods, only k-VERTEX COVER problems for $k \leqslant 200$ were considered solvable [16]. Similar to the sequential wall clock times shown in Fig. 3, we observe that the parallel wall clock times shown in Fig. 5 do not strictly increase with either k or k'. For the "real world" data sets shown, the structure of the graphs is clearly an important factor for the performance of `Code-p`. Most interestingly, the PHD data set with $k = k' = 603$ can be solved in under 10 min, on average.

We executed `Code-p` on 1, 3, 9, and 27 processors of our parallel machine and measured the average relative speedup with respect to parallel wall clock time. As discussed at the end of Section 4.3, the "baseline" (i.e. time for one processor) was set to the minimum of the wall clock times measured for `Code-s` and one processor simulations of `Code-p`. Fig. 6 shows the results for the Somatostatin and WW data sets and Fig. 7 shows the results for the random and grid graphs. Each data point represents the average of 20 experiments. For both cases in Fig. 6, 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" events can occur where the search happens to find a solution near instantaneously.

For Fig. 6, we observe that the average relative speedup grows monotonically in both cases. For the random graph data set in Fig. 6(a), 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 Fig. 6(b), we note that there exist exactly two solutions. As discussed in Section 3.2 and illustrated in Fig. 2, the number of solutions in the search tree is also very important for the

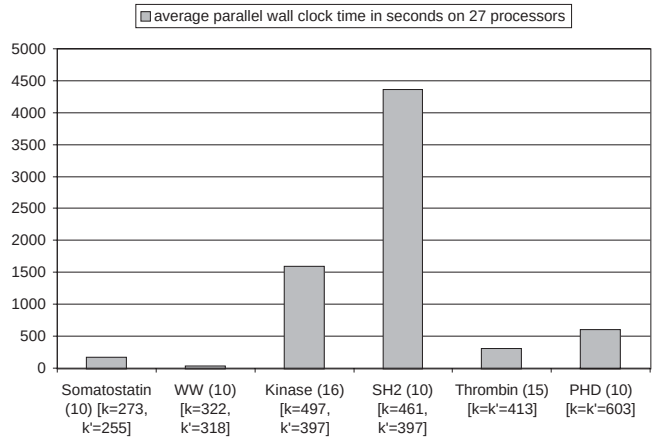

Fig. 5. Average parallel wall clock times.

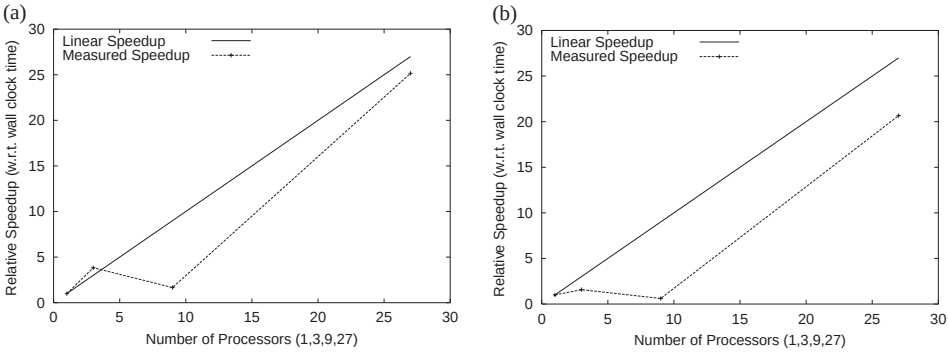

Fig. 6. Average relative speedup for (a) somatostatin and (b) WW.

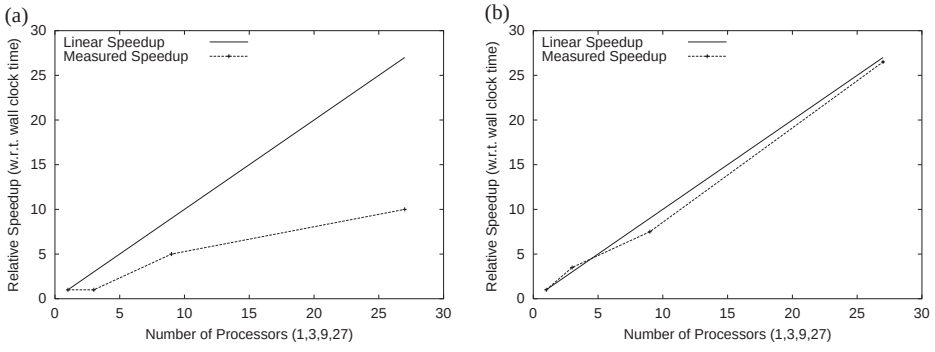

Fig. 7. Average relative speedup for (a) a random graph and (b) a grid graph.

relative speedup. We conjecture that this is the reason why, in Fig. 6(b), the slope of the average relative speedup curve is very close to linear.

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 implemented and tested a new parallel FPT method for the k-Vertex Cover problem and provided the first experimental examination of parallel FPT techniques.

By solving "real world" problem instances with $k \geqslant 400$ in typically less than 1.5 h, our code can handle larger instances of k-Vertex Cover than any previously reported sequential implementation.

Acknowledgments

We thank the referees for their very helpful suggestions on possible improvements of this paper.

References

[1] R. Balasubramanian, M.R. Fellows, V. Raman, An improved fixed-parameter algorithm for vertex cover, Inform. Process. Lett. 65 (1998) 163–168.

[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, Oconomowoc, WI, 1995, pp. I: 11–14.

[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, SIAM J. Comput. 22 (1993) 560–572.

[5] M. Cesati, M. Di Ianni, Parameterized parallel complexity, in: Proceedings of the Fourth International Euro-Par Conference, Southampton, UK, 1998, pp. 892–896.

[6] A. Chan, F. Dehne, A note on coarse-grained parallel integer sorting, in: Proceedings of the 13th Annual International Symposium on High Performance Computers (HPCS'99), Kingston, Canada, 1999, pp. 261–267.

[7] J. Chen, I.A. Kanj, W. Jia, Vertex cover: further observations and further improvements, in: P. Widmayer, G. Neyer, S. Eidenbenz (Eds.), 25th International Workshop on Graph-Theoretical Concepts in Computer Science (WG'99), LNCS, Springer, Berlin, 1999.

[8] F. Dehne, Guest editor's introduction, (special issue on Coarse grained parallel algorithms), Algorithmica 24 (3/4) (1999) 173–176.

[9] F. Dehne, A. Fabri, A. Rau-Chaplin, Scalable parallel computational geometry for coarse grained multicomputers, Internat. J. Comput. Geom. 6 (3) (1996) 379–400.

[10] L. DeVroye, Private communication, GRACO Conference, Fortaleza, Brazil, March 2001.

[11] R.G. Downey, M.R. Fellows, Fixed-parameter tractability and completeness, Congr. Numer. 87 (1992) 161–187.

[12] R.G. Downey, M.R. Fellows, Fixed parameter tractability and completeness I: basic theory, SIAM J. Comput. 24 (1995) 873–921.

[13] R.G. Downey, M.R. Fellows, Fixed parameter tractability and completeness II: completeness for $W[1]$, Theoret. Comput. Sci. A 141 (1995) 109–131.

[14] R.G. Downey, M.R. Fellows, Parameterized computational feasibility, in: P. Clote, J. Remmel (Eds.), Feasible Mathematics II, Birkhauser, Boston, 1995, pp. 219–244.

[15] R.G. Downey, M.R. Fellows, Parameterized Complexity, Springer, Berlin, 1998.

[16] R.G. Downey, M.R. Fellows, U. Stege, Parameterized complexity: a framework for systematically confronting computational intractability, in: F. Roberts, J. Kratochvil, J. Nesetril (Eds.), Contemporary Trends in Discrete Mathematics: From DIMACS and DIMATIA to the Future, AMS-DIMACS Proceedings Series, Vol. 49, Amer. Math. Soc. Providence, RI, 1999, pp. 49–99.

[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, New York, 1979.

[19] G. Gonnet, Vertex cover approximation algorithm, http://cbrg.inf.ethz.ch/Server/VertexCover.html, 1999.

[20] R. Niedermeier, P. Rossmanith, Upper bounds for vertex cover further improved, in: C. Meinel, S. Tison (Eds.), Proceedings of the 16th Symposium on Theoretical Aspects in Computer Science (STACS'99), LNCS, Springer, New York, 1999.

[21] C.H. Papadimitriou, M. Yannakakis, On limited nondeterminism and the complexity of the V-C dimension, J. Comput. System Sci. 53 (1996) 161–170.

[22] V.N. Rao, V. Kumar, On the efficiency of parallel backtracking, IEEE Trans. Parallel Distrib. Systems 4 (4) (1993) 427–437.

[23] C. Roth-Korostensky, Algorithms for building multiple sequence alignments and evolutionary trees, Ph.D. Thesis, ETH Zürich, Institute of Scientific Computing, February 2000.

[24] 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.

[25] U. Stege, Resolving conflicts from problems in computational biology, Ph.D. Thesis, No.13364, ETH Zürich, 2000.
[26] J.D. Thompson, D.G. Higgins, T.J. Gibson, CLUSTAL W: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, positions-specific gap penalties and weight matrix choice, Nucleic Acids Res. 22 (1994) 4673–4680.
[27] L. Valiant, A bridging model for parallel computation, Comm. ACM 33 (8) (1990) 103–111.

Further reading

C. Korostensky, U. Stege, G.H. Gonnet, Algorithms for improving multiple sequence alignments and building evolutionary trees, Technical Report, Vol. 321, Inst. of Scientific Computing, ETH Zuerich, 1999.
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.