Coarse Grained Parallel Fixed-Parameter Tractable Algorithms [*]

– Draft Version –

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

Abstract

FPT algorithms have been successful in allowing to solve *NP*-complete problems for certain problem instances of practical importance. 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 introduce a new definition of ***practical parallel FPT algorithms*** which is based on the speedup obtained for the entire algorithm and addresses shortcomings with previous definitions (*PNC* and *FPP*) that were not efficient in practice. 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 k-Vertex Cover algorithm in C/MPI 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

Fixed-parameter tractability (*FPT*) has been proposed as a means of confronting the obstacle of *NP*-Completeness [10, 11, 12, 13, 14, 15, 16]. 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, 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. $f(k) = 2^k$).

[*]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.tuns.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.

The two fundamental algorithmic techniques for solving *FPT* problems are *kernelization* and *bounded tree search*. 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. Section 2.2 discusses this approach in detail for the k-VERTEX COVER problem.

FPT algorithms have been successful in allowing to solve *NP*-complete problems for certain problem instances of practical importance which were not feasible with previous methods [10]. The goal of our research is to enhance this approach through the addition of parallelism, thereby allowing even larger problem instances to be solved in practice.

In [3, 8], first definitions were formulated for efficiently parallelizable parameterized problems (*PNC* and *FPP*; see Section 2.3). 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 $4 \log n + \mathcal{O}(k^k)$, using n^2 processors, was presented in [8]. Unfortunately, the practical reality of *FPT* algorithms is rather different. Typical sequential *FPT* algorithms spend minutes on the kernelization phase and hours on the bounded tree search. All previous approaches [3, 8] 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 introduce a new definition of *practical parallel FPT algorithms* which is based on the speedup obtained for the entire algorithm (Section 3). We present a general methodology for parallelizing the bounded tree search phase of *FPT* algorithms.

In this paper, we apply our methodology to the k-VERTEX COVER problem which has important applications, e.g., in multiple sequence alignments for computational biochemistry (see Section 2.2). In fact, for ease of presentation, we introduce our tree search parallelization method by describing its application to the k-VERTEX COVER problem (Section 4). 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 [9]) and BSP (Bulk-Synchronous Parallel [30]) machine models. 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. Note that, the previous results [3, 8] apply to the theoretical PRAM model only.

We have implemented our parallel k-VERTEX COVER algorithm in C/MPI and tested it on a network of 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.

In [10], the authors state that the largest parameter for which the k-VERTEX COVER problem has been solved is $k = 60$, and they note that the only constraint on the algorithm for larger k is that it would fail to terminate in reasonable time. An anecdotal report is given that solving a single k-VERTEX COVER instance for $k \approx 157$ requires a period of 24 hours. In contrast, we have solved a k-VERTEX COVER instance for $k = 200$, $|V| = 900$, $|E| = 50000$ in 30 minutes, using 243 virtual processors on 10 physical processors of a

SUN Ultra SPARC network. A detailed experimental analysis of our algorithm is given in Section 4.4.

The remainder of this paper is organized as follows. Section 2 reviews the definition of fixed parameter tractability, previous results on the k-Vertex Cover problem, and the previous parallel parameterized complexity classes *PNC* and *FPP*. Section 3 introduces our new, more practical, definition of parallel *FPT* which we refer to as FPT^p. Section 4 presents our main result, a coarse grained parallel algorithm for the k-Vertex Cover problem. This algorithm also introduces our new tree search parallelization which can be easily generalized to other *FPT* algorithms that use kernelization and bounded tree search. Section 4.1 presents our parallel kernelization method, Section 4.2 presents our parallel tree search method, and Section 4.3 outlines a preliminary performance analysis through simulation. Section 4.4 discusses our C/MPI implementation of our parallel k-Vertex Cover algorithm, and detailed performance results for a network of Sun SPARC workstations.

2 Review

2.1 Fixed Parameter Tractability (FPT)

Fixed-parameter tractability (*FPT*) has been proposed in [10, 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 [10] 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* [10], 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.

2.2 k-Vertex Cover

We now discuss the k-Vertex Cover problem which has been shown to be fixed parameter tractable and which has important applications, e.g., in computational biochemistry [29]. 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 has important applications in multiple sequence alignments for computational biochemistry [29]. 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 [10, 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 $\mathcal{O}(k^2)$.

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 $N(v) \subseteq VC$ [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 $\mathcal{O}(2^k|V|)$ whether G has a vertex cover of size k.

The first Vertex Cover algorithm is due to Buss and has an $\mathcal{O}(kn + 2^k k^{2k+2})$ time complexity [4]. Papadimitriou and Yannakakis, while proving that k-Vertex Cover $\in P$ when k is restricted to $\mathcal{O}(\log n)$, provided an $\mathcal{O}(3^k n)$ algorithm using maximal matchings [26]. Downey and Fellows presented a different algorithm that runs in time $\mathcal{O}(kn + 2^k k^2)$ [14]. Balasubramanian, Fellows, and Raman suggested two different FPT algorithms in their publication [1]. The running times of the algorithms are $\mathcal{O}((\sqrt{3})^k k^2 + kn)$ and $\mathcal{O}((1.324718)^k k^2 + kn)$, respectively. The first of these two algorithms will form the basis of the parallel algorithm described later 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 $\mathcal{O}(kn + r^k k^2)$, $r \approx 1.3195$ [10]. An algorithm by Niedermeier and Rossmanith runs in time $\mathcal{O}(kn + r^k k^2)$, $r \approx 1.2917$, using an improved search tree [24]. Recently, Stege combined the results of [10], [24] and, using an improved kernelization and search tree, developed an algorithm with running time of $\mathcal{O}(kn + r^k k)$, $r \approx 1.2906$ [28, 29]. A further improvement was made in [7], where the running time was reduced to $\mathcal{O}(kn + 1.271^k k^2)$.

2.3 Parallel Parameterized Complexity Classes 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 [8]. They also propose the first FPT EREW-PRAM algorithm for solving the k-Vertex Cover problem in time $4\log n + \mathcal{O}(k^k)$, 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 $\mathcal{O}(\log^k n)$ time, using $n^{\mathcal{O}(1)}$ processors, where n is the length of the input, and k is a constant independent of n [22]. 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.

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

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 [8] 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.

3 Parallelizable FPT: FPT^p

The definitions of *FPP* and *PNC* in [3, 8] 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 [8] has running time $4\log n + \mathcal{O}(k^k)$, 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 on the bounded tree search. The approach in [3, 8] 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, 8] 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 [30], CGM [9], and LogP [23], 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., $\alpha =$ 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 time T_p for p processors such that $T_1 = f(k)\cdot n^{\beta}$ and $T_p = \mathcal{O}((\frac{T_1}{p}))$.

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, $FPT^p_{CGM} \subset FPT^p_{BSP} \subset FPT^p_{LogP} \subset FPT^p_{PRAM}$. For the remainder of this paper, we define $FPT^p = FPT^p_{CGM}$.

All algorithms presented in the remainder will be for the CGM model. A coarse grained multiprocessor simply consists of p processors, P_0, P_1, …, P_{p-1}, connected via any communication network or shared memory. Each processor has $\mathcal{O}((n/p))$ local memory. These assumptions are minimal and include most commercially available parallel machines, in particular Beowulf-type cluster machines [2] and networks of workstations. Consult [30, 9] for more details.

4 A Coarse Grained Parallel k-Vertex Cover Algorithm

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

4.1 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 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 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 $\mathcal{O}((1))$ parallel integer sorts which can be implemented via deterministic sample sort [6]. Note that other kernelization rules can be applied as described in [10] and [1]. These rules are also easily reduced to $\mathcal{O}((1))$ 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 $\mathcal{O}((1))$ parallel integer sorts, using deterministic integer sample sort [6].
(1.2) Output either a kernelized graph $\langle G' = (V', E'), k'\rangle$, or VC ($\leq k$), or *"No"*.
— End of Algorithm —

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

4.2 Parallel 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:

[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.

(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 $\mathcal{O}((\log p))$ 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. What speedup is obtained through this parallel exploration of subtrees? After all, only one solution needs to be found. Consider the path Λ in which the sequential algorithms 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}.

4.3 Performance Analysis: 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 over the search path Λ. For our experiment, we build an array of p rows and 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 $l_{par} = \min_{1 \leq i \leq p} l_i$.

Results are shown in Figure 1. The experiments were performed for $L = 1,000,000$, $m = 1, 10$, 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. We ran the experiment for many other combinations of L, m, and p, and the results were always very similar. This observation is very encouraging. We conjecture that the expected value of l_{seq}/l_{par} is indeed p and are currently engaged in formulating a mathematical proof for this.

We also observed that a uniform distribution of the m solutions over the array does not constitute a *best case* scenario. On the contrary, when solutions where non-uniformly distributed, the processor whose search path starts close to a cluster had a high probability of finding a solution much faster than in the uniform case.

4.4 Performance Analysis: Full Implementation

We have implemented our coarse grained parallel k-VERTEX COVER algorithm presented above in C/MPI and tested it 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. 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. The workstation cluster forms the backbone of the graduate student computing facilities, and so experiments were 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 two different sets of test data, representing two extreme classes of graphs. Due to resource limitations, certain constraints were necessary to achieve results in reasonable time. The range of the vertex cover sizes then represent a compromise between the overall experimental runtimes and interesting performance disparity.

Given a graph instance, each experiment was repeated 20 times, for $p = 1, 3, 9, 27, 81, 243$. The runtimes used to calculate the speedup consist of averages over the 20 runs.

Test Data Set 1

We implemented our own graph generator which is able to produce two types of graphs: (1) *Random graphs* which possess little discernible structure and where vertex cover nodes have relatively high degree, and (2) *Grid graphs* which are highly structured and where vertex cover nodes are indistinguishable from non-cover nodes. The performance results shown in this paper are for the following graph instances:
Random graphs:

RG.1 (Figure 2): $|V| = 700$, $|E| = 1000$, $|VC| = 32$
RG.2 (Figure 3): $|V| = 700$, $|E| = 1000$, $|VC| = 40$
Grid graphs:
GG.1 (Figure 4): $|V| = 64$, $|E| = 112$, $|VC| = 32$
GG.2 (Figure 5): $|V| = 81$, $|E| = 144$, $|VC| = 40$

Test Data Set 2

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 heuristic for computational biology research[21]. The performance results shown in this paper are for the following graph instances selected from Gonnet's data set:
G.203 (Figure 6): $|V| = 60$, $|E| = 246$, $|VC| = 41$
G.205 (Figure 9): $|V| = 60$, $|E| = 246$, $|VC| = 41$
G.293 (Figure 10): $|V| = 62$, $|E| = 256$, $|VC| = 43$
G.299 (Figure 11): $|V| = 65$, $|E| = 272$, $|VC| = 43$
G.300 (Figure 12): $|V| = 65$, $|E| = 272$, $|VC| = 45$

Discussion of Test Results

First we must justify the constraint of averaging 20 runs per instance, per processor. In order to establish this as a reasonable restriction, we subjected the graphs G.203 and G.205 to 20 and 100 repetitions in separate experiments. The graphs in 6, 7, and 9, 8 represent the speedup measured. Note that the graphs exhibit similar characteristics. From this we concluded that 20 repetitions was enough to capture the significance of the experiments.

As observed in the graphs in Figures 2, 4, 3, 5, the algorithm achieves linear or super-linear speedup for sufficiently large vertex cover instances. The performance of the algorithm on the second test data set is rather striking, as observed in the graphs in Figures 6, 9, 10, 11, and 12. We see even more dramatically the impact of p processors.

When considered from a practical standpoint (in the FPT^p context), the search tree phase can be considered highly efficient.

There exists a body of research regarding observed speedup anomalies, particularly in the context of discrete optimization problems (e.g., [20], [19]). In a forthcoming version of this paper, we will analyze the observed speedup phenomena in the context of a probabilistic model.

5 Conclusion and Future Work

This paper represents a preliminary effort to establish the FPT approach in a practical parallel context. We introduce the first parallel CGM algorithm for the k-Vertex Cover problem. The experimental results clearly demonstrate the potential of combining parallel computation with FPT techniques.

Although we ran the tests for a small numbers of graphs, they were representative of two extremes of problem instances. Clearly, there remains more testing to be done for larger and more varied input.

In a forthcoming version of this paper, we will discuss the observed speedup phenomena in the context of a probabilistic model.

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 and J. Goldsmith. "Nondeterminism within P". In *SIAM Journal of Computing*, Vol.22, pp. 560–572, 1993.

[5] L. Cai, J. Chen, R.G. Downey and M.R. Fellows. "On the Parameterized Complexity of Short Computation and Factorization". In *Arch. for Math. Logic*, Vol.36, pp. 321–337, 1997.

[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, and W. Jia. "Vertex Cover: Further observations and further improvements". In *25th International Workshop on Graph-Theoretical Concepts in Computer Science (WG'99)*, LNCS, 1999.

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

[9] 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.

[10] 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.

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

[12] R.G. Downey, M.R. Fellows, U. Stege. "Computational Tractability: A View from Mars". In *Bulletin of the EATCS*, No.69, pp. 73–97, October 1999.

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

[16] R.G. Downey and 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] A. Grama, V. Kumar, P. Pardalos. “Parallel Computing of Discrete Optimization Problems”. In *Encyclopedia of Microcomputers*, Marcel Dekker Publishers, NY, 1992.

[20] A. Grama, V. Kumar. “Parallel Processing of Combinatorial Optimization Problems”. In *ORSA Journal of Computing*, 1995.

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

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

[23] D.E.Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, T. von Eicken. “LogP:Towards a Realistic Model of Parallel Computation”. In *4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[24] R. Niedermeier and 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.

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

[26] 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.

[27] D.R. Smith. “Random Trees and the Analysis of Branch and Bound Procedures”. In *Journal of the Association of Computing Machinery*, Vol.31, pp. 163–188, 1984.

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

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

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

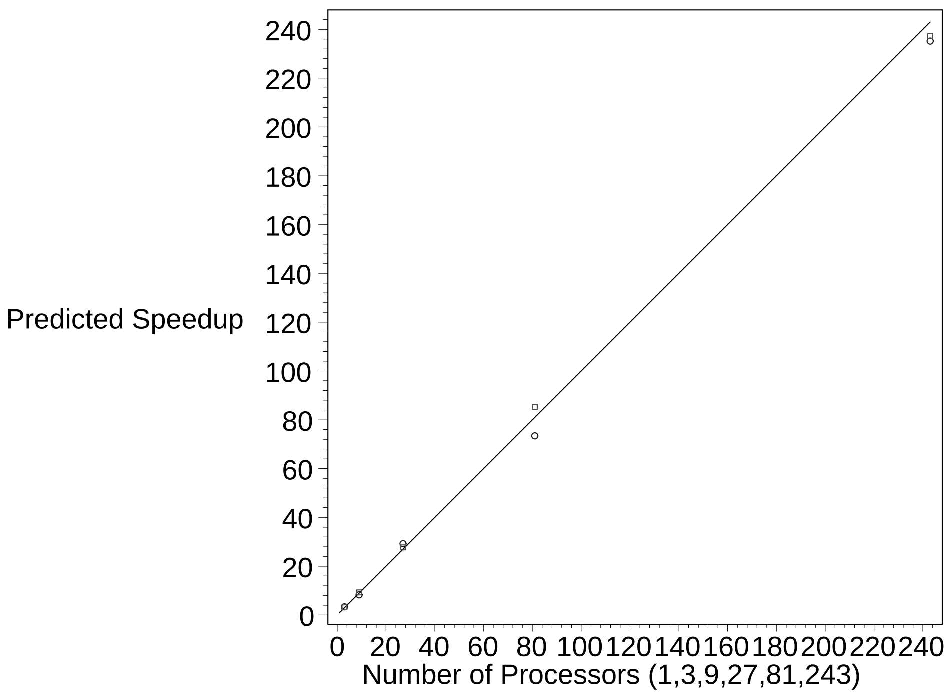

Figure 1: Simulated Speedup Estimation Through “Balls in Bins” Experiment.

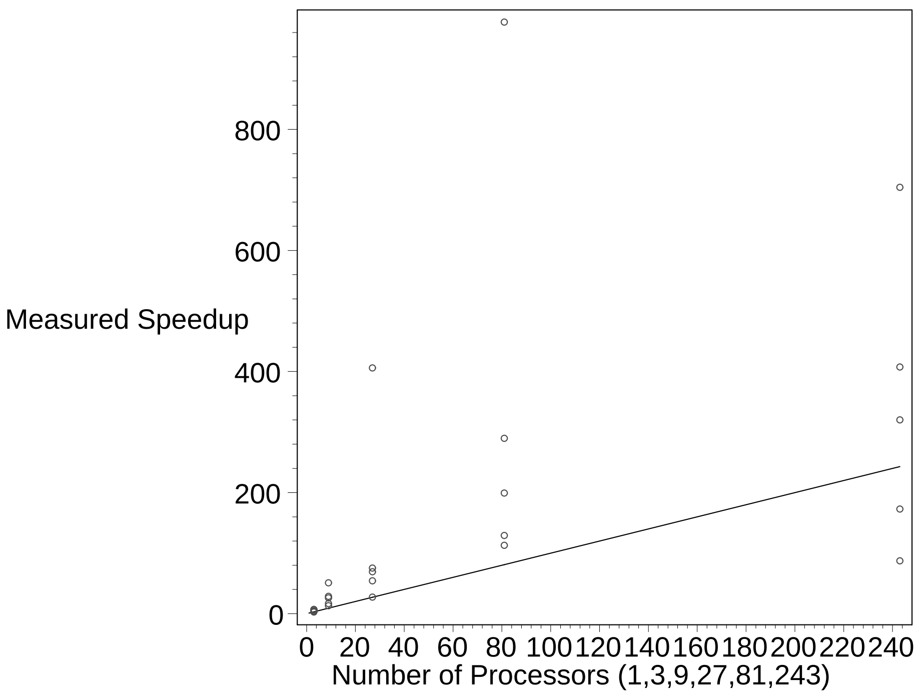

Figure 2: Average speedup measured for random graph RG.1 ($|V| = 700$, $|E| = 1000$, $|VC| = 32$, 20 experiments per data point).

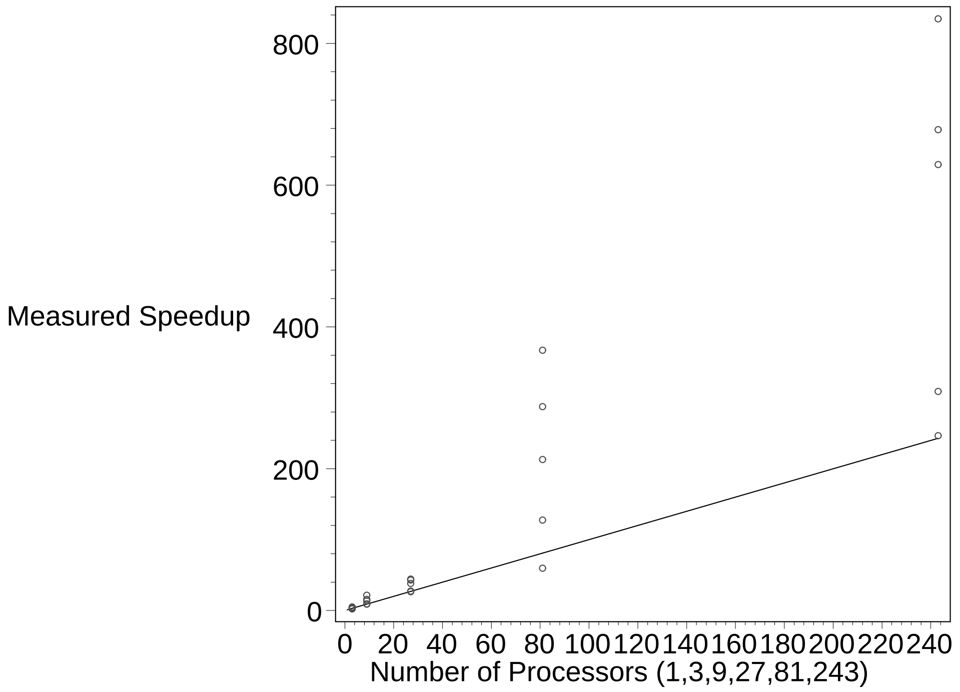

Figure 3: Average speedup measured for random graph RG.2 ($|V| = 700$, $|E| = 1000$, $|VC| = 40$, 20 experiments per data point).

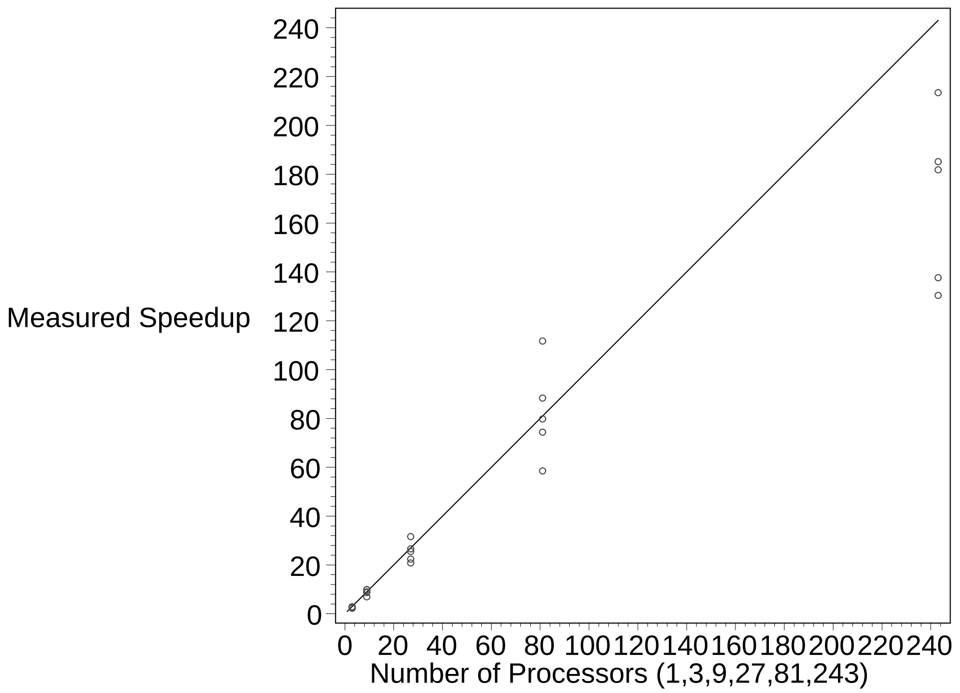

Figure 4: Average speedup measured for grid graph GG.1 ($|V| = 64$, $|E| = 112$, $|VC| = 32$, 20 experiments per data point).

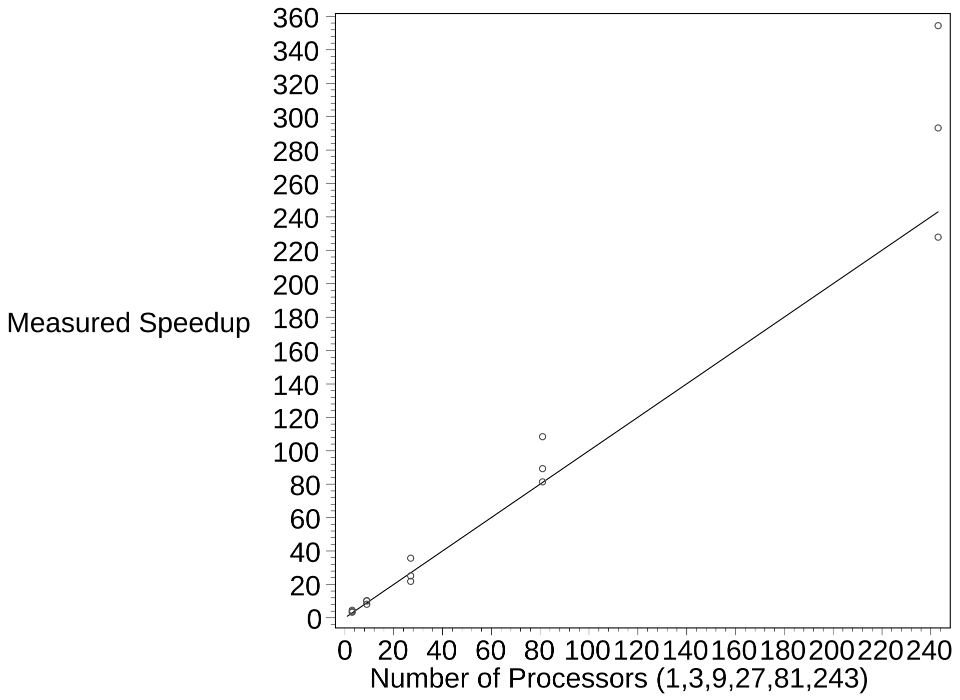

Figure 5: Average speedup measured for grid graph GG.2 ($|V| = 81$, $|E| = 144$, $|VC| = 40$, 20 experiments per data point).

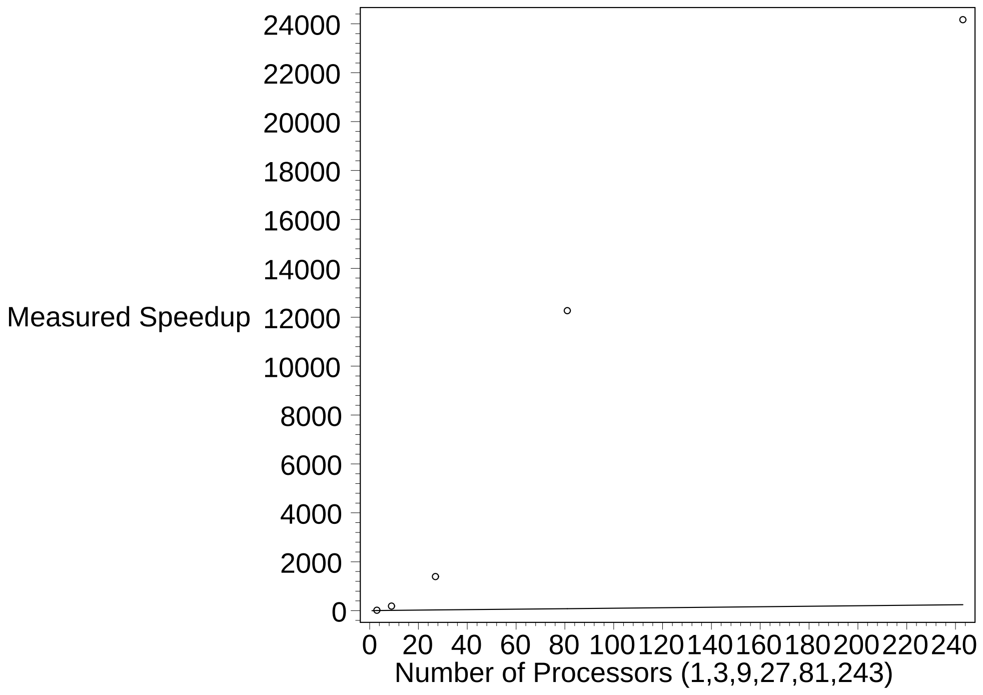

Figure 6: Average speedup measured for Gonnet's graph G.203 ($|V| = 60$, $|E| = 246$, $|VC| = 41$, 20 experiments per data point).

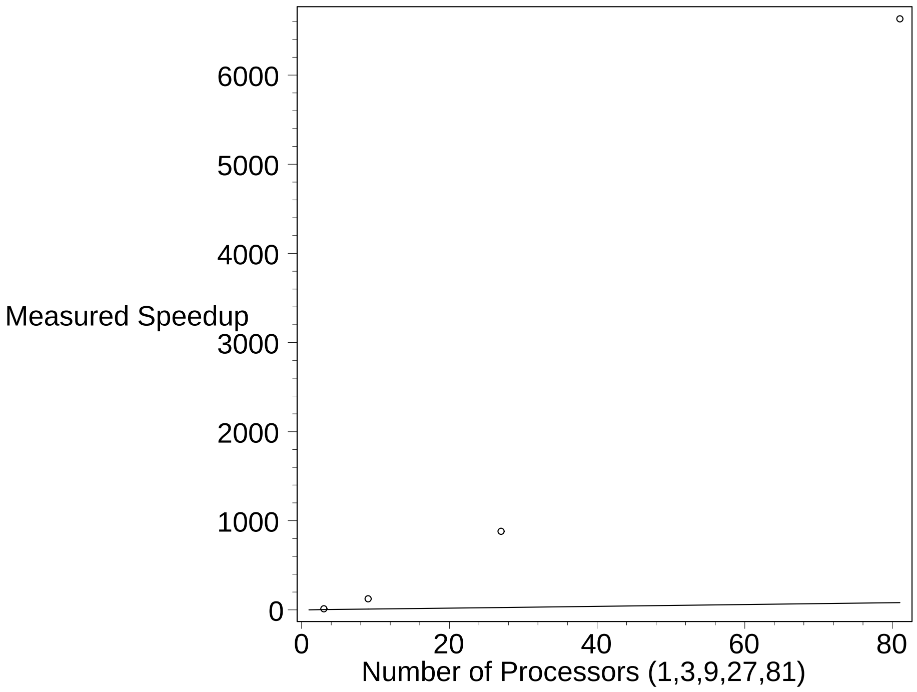

Figure 7: Average speedup measured for Gonnet's graph G.203 ($|V| = 60$, $|E| = 246$, $|VC| = 41$, 100 experiments per data point).

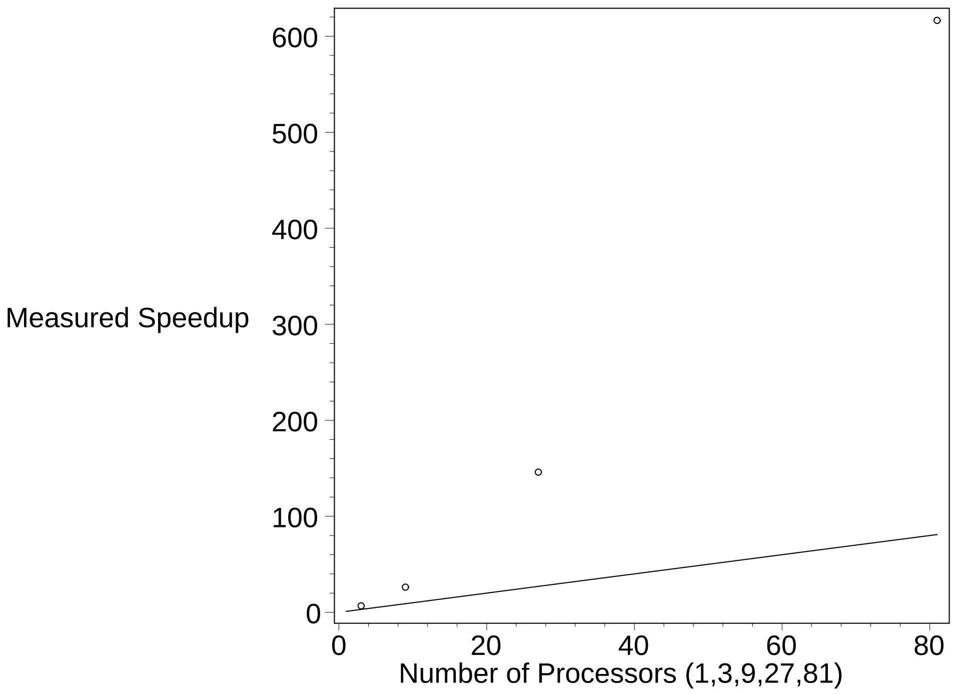

Figure 8: Average speedup measured for Gonnet's graph G.205 ($|V| = 60$, $|E| = 246$, $|VC| = 41$, 100 experiments per data point).

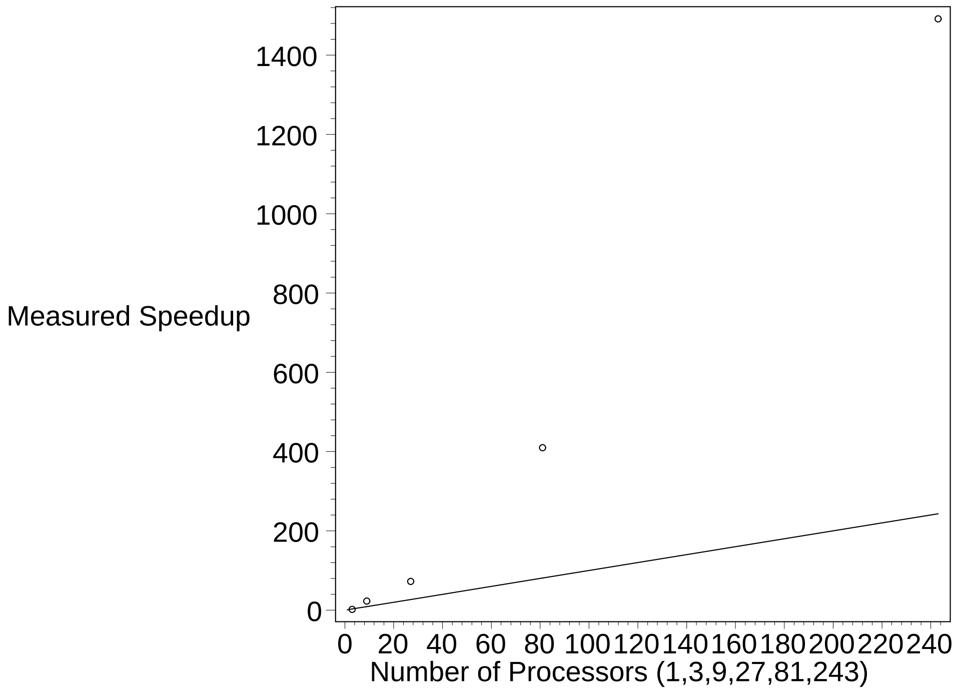

Figure 9: Average speedup measured for Gonnet's graph G.205 ($|V| = 60$, $|E| = 246$, $|VC| = 41$, 20 experiments per data point).

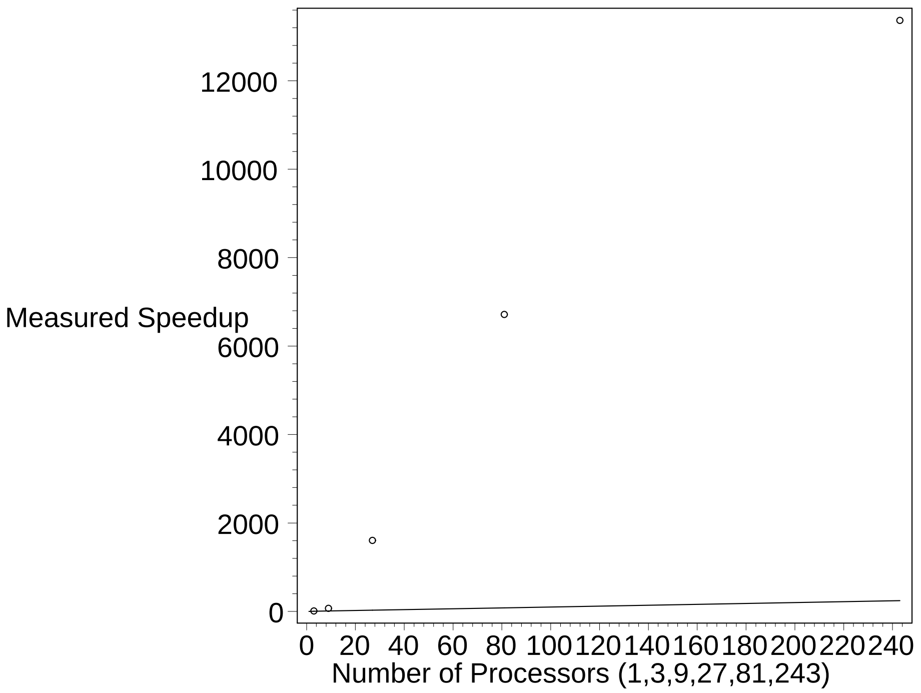

Figure 10: Average speedup measured for Gonnet's graph G.293 ($|V| = 62$, $|E| = 256$, $|VC| = 43$, 20 experiments per data point).

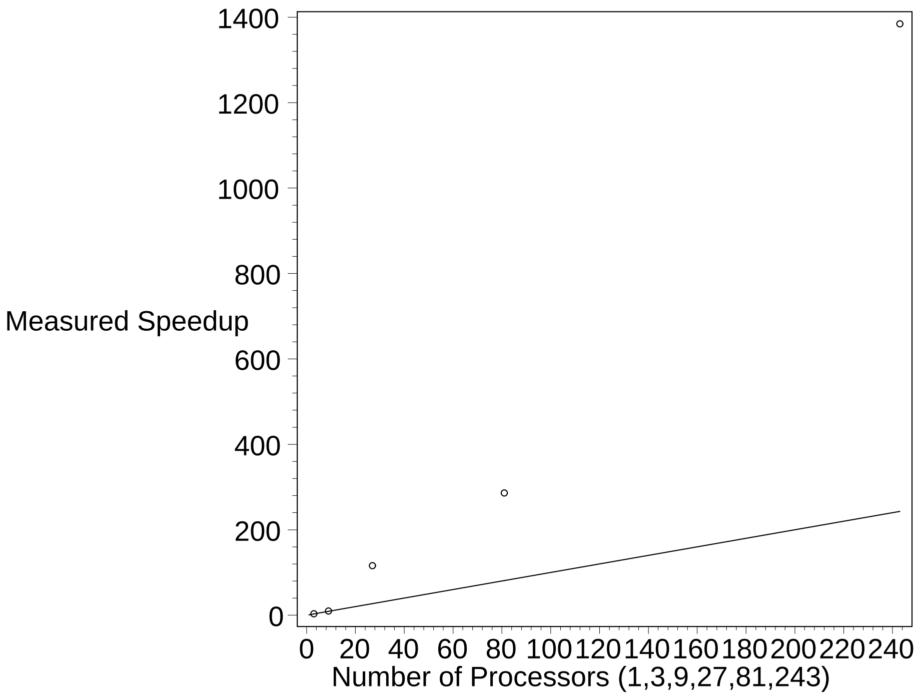

Figure 11: Average speedup measured for Gonnet's graph G.299 ($|V| = 65$, $|E| = 272$, $|VC| = 43$, 20 experiments per data point).

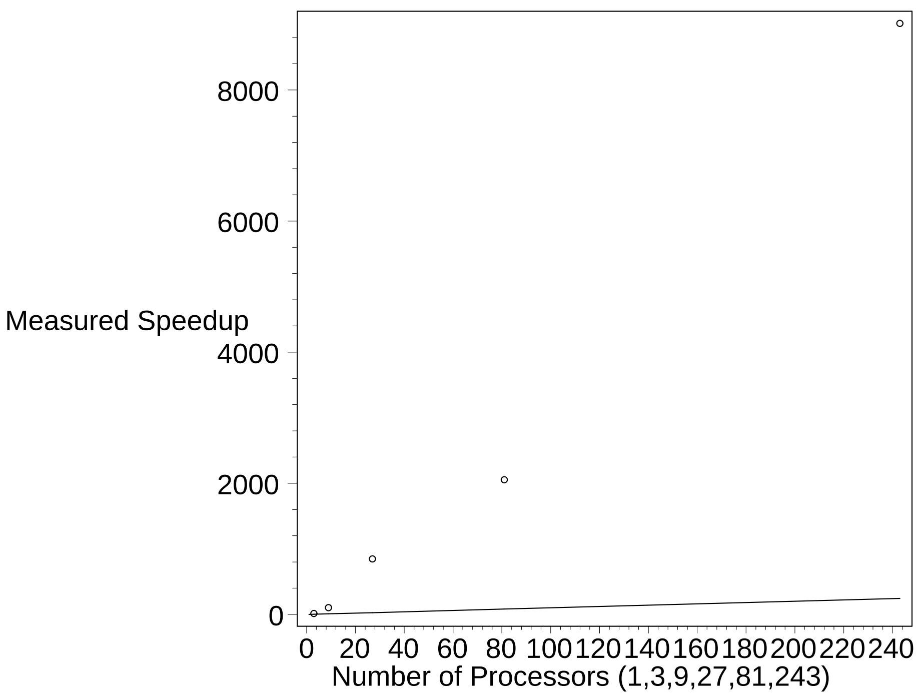

Figure 12: Average speedup measured for Gonnet's graph G.300 ($|V| = 65$, $|E| = 272$, $|VC| = 45$, 20 experiments per data point).