Scalable Parallel Algorithms for FPT Problems[*][†]

Faisal N. Abu-Khzam, Michael A. Langston, Pushkar Shanbhag and Christopher T. Symons
Department of Computer Science, University of Tennessee, Knoxville, TN 37996–3450

Abstract

Algorithmic methods based on the theory of fixed-parameter tractability are combined with powerful computational platforms to launch systematic attacks on combinatorial problems of significance. As a case study, optimal solutions to very large instances of the $\mathcal{NP}$-hard vertex cover problem are computed. To accomplish this, an efficient sequential algorithm and various forms of parallel algorithms are devised, implemented and compared. The importance of maintaining a balanced decomposition of the search space is shown to be critical to achieving scalability. Target problems need only be amenable to reduction and decomposition. Applications in high throughput computational biology are also discussed.

Key Words

Algorithm Design, Parallel Computing, Optimization, Load Balancing, Applications

1 Preliminaries

An innovative technique for dealing with foundational $\mathcal{NP}$-complete problems is based on the theory of *fixed-parameter tractability*.

> A problem of size n, parameterized by k, is fixed-parameter tractable if it can be decided in $O(f(k)n^c)$ time, where f is an arbitrary function and c is a constant independent of both n and k.

[*]This research is supported in part by the National Science Foundation under grants EIA–9972889 and CCR–0075792, by the Office of Naval Research under grant N00014–01–1–0608, by the Department of Energy under contract DE–AC05–00OR22725, and by the Tennessee Center for Information Technology Research under award E01–0178–261.

[†]A preliminary version of a portion of this paper was presented at the International Conference on Parallel and Distributed Computing and Systems held in Marina Del Rey, California, in November, 2003.

The origins of fixed-parameter tractability (henceforth FPT) can be traced back nearly twenty years, to the work by Fellows and Langston on applications of well-quasi order theory, the Robertson-Seymour theorems, nonconstructivity, and in particular the minor and immersion orders. See, for example, [17, 18, 19]. Efforts at that time were motivated by the theme that, by fixing or bounding parameters of relevance to the problem at hand, one might be able to exploit a non-uniform measure of algorithmic efficiency. In the intervening years, Downey and Fellows developed the major theoretical basis of FPT [14]. More recently, something of a cottage industry in FPT algorithm design has begun to flourish, with research groups and workshops now held around the world. Despite all this activity, however, the main focus has remained on theoretical issues, especially worst-case bounds, problem restrictions and the $\mathcal{W}$-hierarchy (a fixed-parameter analog of the polynomial hierarchy). Few serious attempts have been made at large-scale practical implementations. A notable exception is the work of Cheetham et al [8].

2 Exemplar

Perhaps the best-known example of an FPT problem, and the one we use as a case study here, is *vertex cover.* In this problem, the inputs are an undirected graph G with n vertices, and a parameter $k < n$. The question asked is whether G contains a set C of k or fewer vertices that covers every edge in G, where an edge is said to be covered if either (or both) of its endpoints is in C.

In terms of worst-case analysis, the asymptotically-fastest algorithm currently known for vertex cover is due to the work of Chen et al [10], and runs in $O(1.2852^k + kn)$ time. Compare this with $O(n^k)$, the time required to examine all subsets of size k by brute force. Of course an attractive worst-case bound is no guarantee of a practical algorithm. Nevertheless, it is remarkable that the requisite exponential growth (assuming $\mathcal{P} \neq \mathcal{NP}$) has been reduced to a mere additive term.

Algorithms designed to solve FPT problems are sometimes rather loosely termed "fixed-parameter algorithms." Such algorithms were originally intended to work only when the parameter in question was truly fixed. The algorithm described in [7], for example, was aimed solely at determining whether an input graph has a vertex cover of size at most 5.

In contrast, our interest here is on pushing the boundary of feasible computation. We seek to construct effective methods for finding optimal vertex covers in huge graphs, irrespective of any particular parameter value. To accomplish this, we exploit, build upon and implement techniques gleaned in large part from recent advances in the theory of fixed-parameter algorithm design. We detail the salient features of some of these techniques in the next section.

3 Applications

The practical applications of vertex cover are many and varied. A timely topic centers on maintaining security in a computational or communications network. For example, finding a smallest cover may be interpreted as identifying an optimal set of vertices that can control or monitor every transmission link in such a network. Cast in this light, vertex cover is sometimes employed in route-based filtering, which is a popular method for attempting to prevent IP spoofing and distributed denial of service attacks [27].

Much of our recent work, however, has been motivated by ongoing collaborations with mammalian geneticists and neuroscientists. Therefore, with access to large classes of genomic and proteomic data, we shall concentrate here mainly on applications relevant to computational biology. In this environment a great many combinatorial problems hinge on *clique*. In the clique problem, the inputs are an undirected graph G with n vertices, and a parameter $k < n$. The question asked is whether G contains a set C of k or more vertices such that every pair of elements in C is connected by an edge in G.

Clique is not FPT unless the $\mathcal{W}$ hierarchy collapses [14]. Fortunately, vertex cover is a complementary dual to clique. To see this, suppose we wish to determine whether G contains a large clique, where large means of size at least $n - k$ for some suitable choice of k. Let $\overline{G}$ denote the complement of G. Then G has a clique of size at least $n - k$ if and only if $\overline{G}$ has a vertex cover of size at most k. This duality is depicted in Figure 1. Figure 1(a) shows a clique in a sample graph, G_1. Figure 1(b) shows the corresponding vertex cover in $G_2 = \overline{G_1}$.

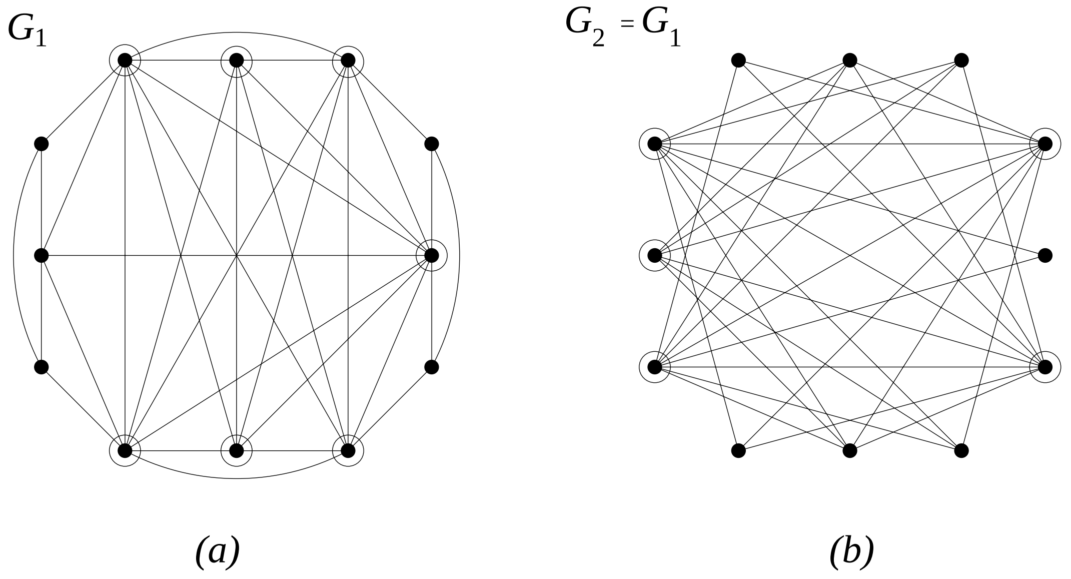

Figure 1: The duality between clique and vertex cover.

One application lies in phylogeny. Here we use protein sequence data, which is now widely available from NCBI, Swiss-Prot and a variety of other sources. Each pair of sequences is assigned a correlation score using codes such as those available in the well-known ClustalW package. A complete edge-weighted graph is then constructed, with vertices representing sequences, and edge weights denoting the corresponding correlation scores. Because source data often contains errors, outliers, duplicates and so forth, we seek to obtain the largest possible set of closely-related sequences before proceeding with the phylogenetic analysis. To accomplish this, edges whose weights fall below some preset threshold are removed; weights on edges that remain are henceforth ignored. It remains to find in the reduced, unweighted graph a largest set of vertices with all possible edges between them. Of course this is just a restatement of the maximization version of the clique problem.

Clique has also proven useful in the analysis of microarray data. With standard statistical tools, raw data is transformed into a correlation matrix, from which we extract a complete, edge-weighted graph. In this graph, vertices represent genes, and edge weights denote estimated correlation values. As in the phylogeny application, edges are eliminated based on some preset threshold. Now solving clique on the reduced, unweighted graph yields a largest set of putatively co-regulated genes. This set can be useful in a variety of projects, a prime example being the quest to understand mechanisms of gene regulatory networks.

Yet another application centers on gene motif discovery. Here the goal is to characterize and annotate collections of genes by finding and matching *cis*-regulatory elements, which are upstream DNA subsequences usually thought to contain on the order of 8-16 base pairs. See [5]. These are just a few examples. There are many others. One needs only to search the literature, via pubmed [24] for instance, to find hordes of clique applications in the biological realm alone.

4 Problem Reduction

The goal of problem reduction is to condense an arbitrary input instance down to a relatively small computational core. The idea is to find such a core whose size depends only on k, and to find it in time polynomial in n. In the context of FPT, this operation is termed "kernelization," with the compute core called the "kernel." Kernelization is most often accomplished with a variety of preprocessing rules. Many of these rules are folklore. Others have been formalized in the literature [6, 10].

For example, for vertex cover it is possible with preprocessing to eliminate vertices of very low or high degree. It is trivial to eliminate vertices of degree one, because there is no gain in using a leaf to cover its only incident edge. It is easy to eliminate vertices of degree two [10]. There is also a rule for handling some but not all vertices of degree three, but it is complicated and not necessarily worth the extra effort in practice. If a vertex has degree $k + 1$ or more, then it must be in any satisfying cover. Otherwise, all its neighbors would be required to be in the cover, and there are simply too many of them. Therefore we can eliminate it and reduce k by one. There is also a rule for handling vertices of degree exactly k. When no more vertices can be eliminated in this fashion, the reduced graph (kernel) has size at most $k^2/3 + k$ [3]. We have implemented these and several other preprocessing rules. The order in which they are performed can sometimes have an impact on the size of the kernel produced. Their most important attributes are probably their speed and simplicity.

Newer and more powerful (but also slower and more complicated) kernelization methods rely on linear programming relaxation and related techniques [22, 23, 25]. We have implemented and fine-tuned these and alternate methods of our own design. We have incorporated where possible highly-efficient LP dual codes graciously provided to us by Bill Cook at Georgia Tech [11]. The culmination of all this is a suite of routines that are capable of producing a kernel of size at most $2k$ [1]. Unfortunately, ensuring such a linear-sized kernel has until very recently been a fairly slow and cumbersome affair, mainly due to the requirements imposed when solving the LP bottleneck. This situation is now changed. In [3] we conducted large-scale empirical studies of these algorithms and, along with Mike Fellows and others, introduced and analyzed a novel approach dubbed “crown decomposition.” With this new technique, we can produce a kernel of size at most $3k$ in $O(n^{2.5})$ time. Extensive testing has shown that crown decomposition works extremely well in practice, especially when used in conjunction with degree-based rules. It generally runs much faster and produces even smaller kernels than the worst-case bounds would suggest.

5 Decomposition and Search

As soon as reduction is complete, the core (kernel) is ready to be passed to the decomposition stage. The problem now becomes one of exploring the kernel’s search space efficiently. In the parlance of FPT, this is known as “branching.” This is an extremely challenging task. Even though the kernel is now of bounded size, its search space typically contains an exponential number of candidate solutions.

We use an implicit tree structure to organize the search for a satisfying cover. Each internal node of the tree represents a choice. For example, one might make the choice at the root by selecting an arbitrary vertex, v. Then the left subtree denotes the set of all solutions in which v is to be in the cover. The right subtree denotes the set of all solutions in which v is not in the cover. Moving on down the tree, each leaf is a set of k or fewer vertices that may or may not form a valid cover, corresponding to a potential solution. Although effective, this form of decomposition is an exhaustive process to be sure. (This should come as no surprise. After all, the underlying problem is $\mathcal{NP}$-complete.)

Decomposition clearly requires the lion’s share of computational resources. Thus, it is important to note that subtrees at each level can in principle be explored in parallel. The depth of the tree can be at most k. All that needs to be done at a leaf is to check whether the removal of the leaf's candidate solution leaves an edgeless graph (all edges are covered). We shall have much more to say about this in subsequent sections.

Decomposition via the branching process is explicated with the following two figures. Figure 2 depicts a sample graph, G, for which we want to find a vertex cover of size five or less. Figure 3 illustrates the actions taken in a resultant tree search, which is rooted at the vertex labeled 0. At each node of the tree, C denotes the candidate cover and k denotes the maximum number of vertices that may still be added to C. (Therefore $|C| + k = 5$.) In this example, we have favored branching at a node of highest current degree and employed a depth-first search. When no satisfying solution exists, the entire tree must of course be searched. Even when solution(s) do exist, they may not be found until much, or perhaps all, of the tree has been examined. In this particular example, however, a solution is eventually found in the root's leftmost subtree, so that the dotted edges leading to the other subtrees need not be traversed by a sequential algorithm. This is not a property easily exploited by parallel algorithms. As we shall see, parallel algorithms may be very lucky, or very unlucky, as the solution space is decomposed. Furthermore, other portions of the tree created by this simple example reveal a rather counter-intuitive feature of the search process. When branching on some vertex, v, solutions may sometimes be found faster if one favors for exploration the subtree in which v is not in the cover. When v is left unused, all its neighbors must be in the cover and, if the degree of v is high, we may converge much more rapidly toward a solution.

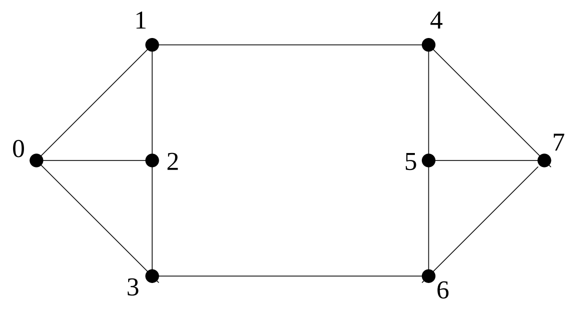

Figure 2: A sample graph, G.

6 Algorithm Synthesis

Reduction and decomposition need not be stand-alone tasks. As decomposition proceeds and new instances are generated, additional reductions in problem size can often be realized by a re-application of preprocessing rules. In the context of FPT, this technique of re-kernelization is often termed "interleaving." See [26] for detailed information and analysis.

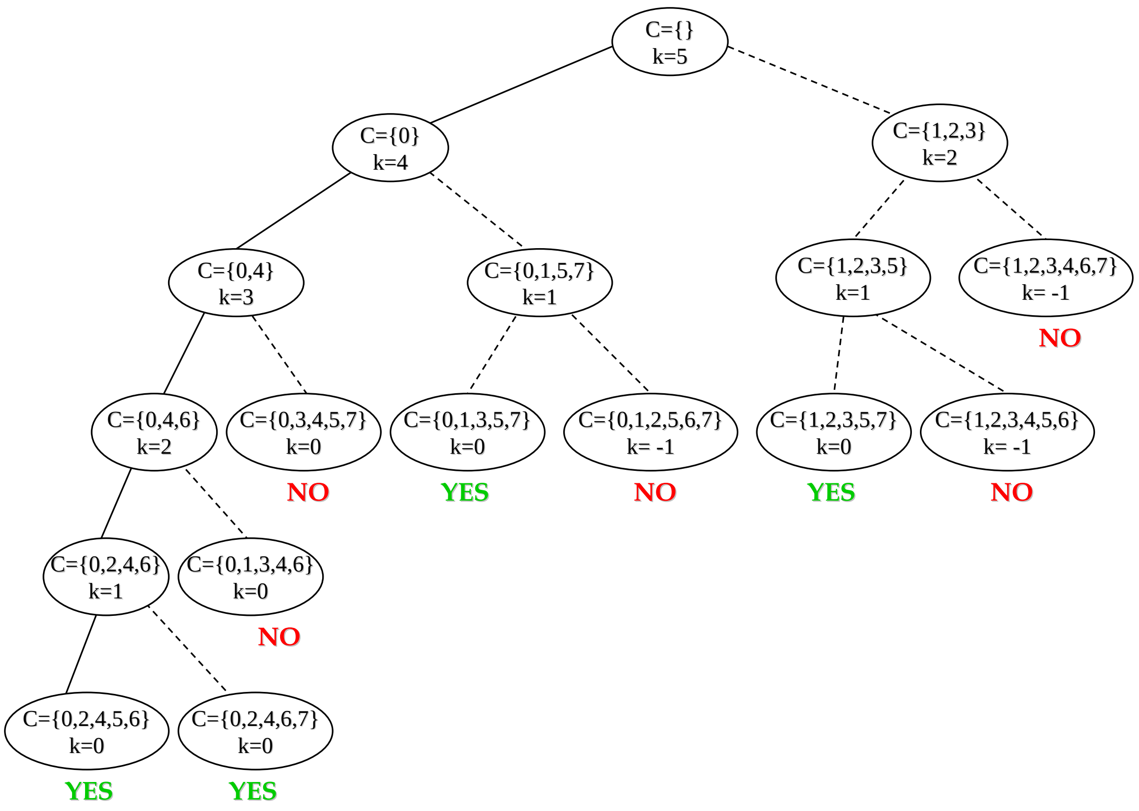

Figure 3: Branching finds a vertex cover for G.

Interleaving's effects can vary greatly, depending on the input. Its impact can sometimes be dramatic, especially on small or sparse instances. On the other hand, we have frequently seen only negligible improvement on large, dense or highly regular graphs. Moreover, the overhead of interleaving can be prohibitive. One must not only re-examine the graph and make changes to it, but also keep track of these changes at each level of the search tree. Thus we use interleaving sparingly.

7 The Case for Parallel Methods

Reduction greatly helps to reduce overall runtimes. In fact it's been established kernelization alone is what makes membership in FPT possible [9, 16]. Nevertheless, decomposition and search remain exhaustive procedures, often incurring enormous computational demands even for problem instances of only modest size. Thus we naturally turn to the use of parallelism. Parallelization works hand-in-hand with the results of decomposition. The task of spawning processes is structured by the same tree that is used to explore the kernel's search space. To explicate, suppose both n and k are large, and 32 processors are available. Because the search tree has a branching factor of two, decomposition will have used the first $5 << k$ levels of its tree to split the input into 32 subgraphs, one for each processor. In turn, each processor will, in parallel, examine its subgraph using the search tree technique.

Once spawned, these tasks are left to run in a virtually unstructured manner. They can be farmed out as the need arises, and serviced in anything from a tightly-coupled to a widely-distributed fashion. No barrier synchronization is needed. No MPI-like tools are required. A process need not even know its siblings exist. Each is free to run to completion, at its own pace, returning its result whenever it is finished. We have run our codes on several different platform/gridware combinations. Our best results have generally been obtained with minimal intervention, however, in the extreme case by directly launching secure shells (SSHs).

This is of course a completely static form of parallel decomposition and load balancing. For all but the smallest values of k, it produces an extremely coarse-grained implementation. If p processors are available, each processor will operate on a distinct subgraph using some parameter value $k' \leq k - \lfloor \log_2 p \rfloor$. Load balancing in this fashion is often good enough but, as we shall see, it is sometimes amazingly ineffective. With no dynamic form of task re-distribution, simple static load balancing can, in this FPT environment, lead to egregious runtimes and dramatic parallel resource under-utilization.

At this time we also perform interleaving in a coarse-grained manner, only when subtasks are spawned, using rules for degrees one, two and $k+1$ or more. Of course interleaving could be applied much more often, in the extreme case at every node in the search tree. After all, the potential utility of re-kernelizing would seem to increase as subgraphs are reduced to manageable sizes during branching. Thus far, however, we have not found the overhead of such a fine-grained form of interleaving to be worth the effort. It is unclear whether some intermediate level of granularity is better suited to problems of interest. We intend to continue studying the effect of varying this and other implementation options.

One additional implementation detail is worth noting. The recursive nature of our algorithms naturally but unfortunately leads to a variety of stack overflow problems. This is particularly true when dealing with very large inputs. To ameliorate this condition, we maintain whenever possible a single global structure for the input graph. Modifications made through branching and interleaving are recorded in a a ternary status vector of length n. A zero indicates that the status of a vertex is still in question; a one indicates that a vertex is in the cover found thus far; a two indicates that a vertex has been deleted. This simple scheme turns out to save a tremendous amount of memory that would otherwise be wasted in copying adjacency matrices from one recursive call to another, and allows us to scale up to much larger problems than would have been possible had we not managed memory more carefully.

8 Initial Results on Synthetic Data

We first tested our algorithms on synthetic data sets. These were mostly pseudo-random graphs generated in a variety of ways. The results were always impressive. In fact some of our best results were obtained on synthetic data sets kindly provided by Frank Dehne at Carleton University [12]. See Table 1. Numbers listed there reflect wall clock times.

These results are intriguing. Three different graphs are listed, each with 600 vertices, and each containing a vertex cover of size 400. On them we used 32 processors, each running at 500 MHz. Note the stunning differences between sequential and parallel decomposition times. At first we thought the sequential routine must be hung in an infinite loop. Traces revealed, however, that it was humming along nicely. It is just that these graphs have a lot of edges and their search spaces are immense. The average speedup we observe is something north of 30,000. Because we are only employing 32 processors, this would surely have to be characterized as super-super-linear!

Graph Name	Sequential Reduction	Sequential Decomposition	Parallel Decomposition
rg30	1 second	halted after two days	5 seconds
rg31	1 second	halted after two days	4 seconds
rg32	1 second	halted after two days	4 seconds

Table 1: Intriguing decomposition times on synthetic graphs.

So should we abandon our streamlined but straightforward sequential code, re-writing it so that it uses multiple threads or otherwise emulates the actions of the parallel version? In order to try to answer this question, we have sought to determine what factor(s) could have caused the unusually fortuitous sort of parallel behavior exemplified by the run times in Table 1.

One such factor is the way in which solutions are scattered about the search space. It has been observed before [8] that solutions tend to be highly non-uniformly distributed. In this setting, therefore, decomposition can do much more for us than merely help guide the search and parallelize the process. One or more processors may find a solution relatively close to the root of its respective subtree. The searches occurring at other processors may be fruitless; it matters not. Our parallel run time is based solely on the time required for the earliest-finishing processor to deliver to us a solution, at which point the other processors are halted. Yet the sequential algorithm is doomed to plod along, exhaustively examining each and every subtree until it stumbles across a solution-laden region of the search space.

Although the scenario just described is plausible enough, surely there is more to the story if we are to understand astonishing speedups such as these. After more digging and tracing, we now believe we have a good handle on the primary factor at play here, namely, the makeup of the data itself. These synthetic graphs turn out to be highly non-random. In fact they are somewhat grid-like which, because of the way we implement branching, places our parallel algorithm at a great advantage. In short, our parallel algorithm was lucky. Thus, the super-super-linear speedups we observe are mainly artifacts of the data. Nevertheless, these results do serve an important purpose. They caution us against trying to read too much into contrived examples, and echo an all-too-familiar story: *rely only on real data*.

9 Dynamic Parallel Decomposition

As we move toward the use of non-synthetic data, the static parallel decomposition technique just outlined has proved useful in many of our experiments. The result has generally been much smaller runtimes than those for corresponding sequential codes. In some nagging cases, however, we would observe only very small, sometimes even negligible, speedups. In an occasional extreme case, all but one of the processors would finish quickly, leaving the lone remaining processor to do the bulk of the computation. In short, we were unlucky. Observe that we can get lucky, achieving excellent speedup, only on “yes” instances. But we can get unlucky, achieving little or no speedup at all, on both “yes” and “no” instances. Interestingly, we have found that as we iterate the decision algorithm to attain optimality, the closer we get to converging the parameter to its optimal value, the more likely we are to get unlucky.

Thus, to maintain scalability as more and more machines come on line, it has been imperative that we incorporate at least some primitive form of dynamic load balancing into our methods. We have studied a number of strategies, but even a simple scheme seems to have a tremendous impact. We hesitate to interrupt active processes, and therefore refrain from redistributing processor loads until all processors but one are idle.

Working with colleagues in proteomics, we have downloaded vast assortments of sequence data against which to test our codes. These have been obtained mainly from the National Center for Biotechnology Information (NCBI). Each data set corresponds to a family of protein sequences that share a common domain. A representative set of results using data from the *sh2* and *sh3* domains is reported in Table 2. As before, we used 32 processors, each running at 500 MHz. Wall clock times are listed. We have chosen to highlight these results because they are particularly telling. In them the relevant parameter is just converging on the optimal value. Note the success in load balancing achieved with dynamic decomposition.

Graph		Cover	Instance	Sequential	Parallel Decomposition	
Name	Size	Size	Type	Decomposition	Static	Dynamic
sh2-5	839	399	yes	7 sec	not needed	not needed
sh2-5	839	398	no	141 min	82 min	20 min
sh3-10	2,466	2,044	yes	just under 5 days	just under 5 days	140 min
sh3-10	2,466	2,043	no	halted after 6 days	halted after 6 days	620 min

Table 2: A comparison of static versus dynamic parallel decomposition times.

10 A Middleware Approach

We have also implemented our codes using NetSolve, a grid middleware system based upon the agent-server-client model [4] and an integral part of the SInRG project pioneered here at the University of Tennessee [13]. See Figure 4.

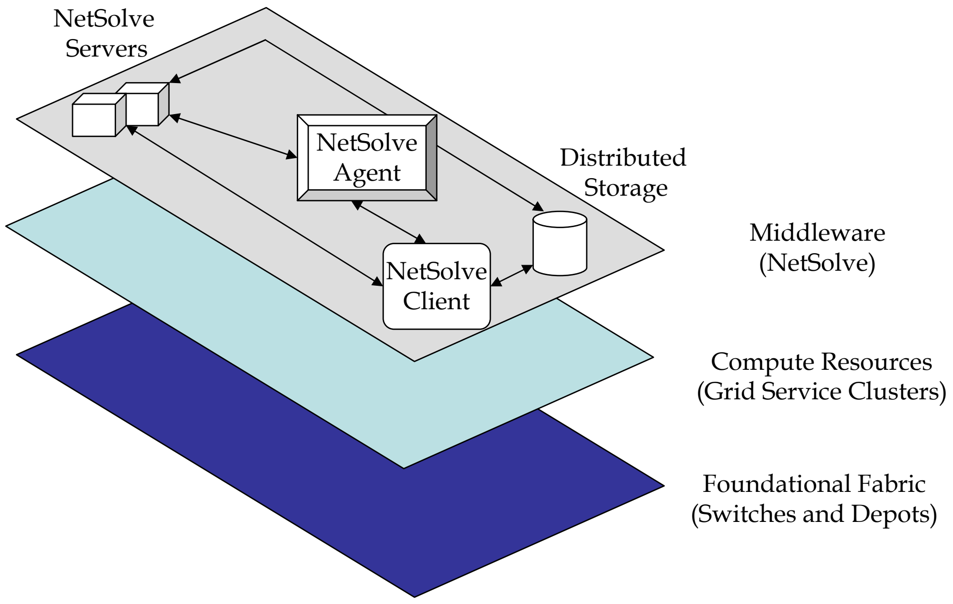

Figure 4: An overview of NetSolve and its role in the SInRG project.

NetSolve implements remote computing, in which the program to be executed is stored on the server, on both isolated systems and grid clusters located around the world. Client-server communication is carried out through standard TCP/IP sockets. The client polls the NetSolve agent in order to locate the server most appropriate for a given job. We utilize NetSolve 2.0, which offers many advantages for our application. For example, when NetSolve assigns jobs to remote processors, its agent takes into account network latencies, computational speeds, and current workloads in order to make an informed choice. Furthermore, NetSolve offers ease of access to computational resources, allowing a wide range of users to run our codes on multiple processors (in contrast, our SSH-based approach requires that users have accounts on each machine utilized). Moreover, NetSolve allows us to operate seamlessly in a heterogeneous computing environment, while offering some level of built-in fault tolerance.

Of course there is a price to be paid for such convenience. With any type of middleware, we much be cognizant of potential overhead issues, many of which can substantially slow down our implementations. Our computational experience suggests that the middleware premium is only moderate. Sample results are displayed in Table 3. Graphs in the "ys" family are relatively sparse, and are produced from yeast-stress microarray data. The remaining graphs are denser, and are derived from the *globin* and *sh2* protein domains, respectively. These experiments were conducted using 32 processors, each running at 500 MHz.

Graph		Cover	Instance	Parallel Decomposition	
Name	Size	Size	Type	Static	Netsolve
ys-7	2,561	2,342	yes	14 sec	16 sec
ys-7	2,561	2,341	no	15 sec	19 sec
globin-15	972	427	yes	14 sec	16 sec
globin-15	972	426	no	18 sec	19 sec
sh2-4	839	337	yes	43 min 13 sec	51 min 11 sec
sh2-4	839	336	no	56 min 49 sec	59 min 27 sec
sh2-10	839	547	yes	5 min 32 sec	6 min 59 sec
sh2-10	839	546	no	6 min 3 sec	8 min 51 sec

Table 3: A comparison of SSH versus NetSolve implementations.

NetSolve uses a "problem description file" to control access to code stored on its servers. Participants must compose and debug these files when implementing routines other than NetSolve's built-in functions. Problem description files are not well-known and sometimes rather tricky to get just right. Therefore, as an aid to readers and potential users, we list in an appendix the problem description file for our parallel decomposition method.

11 Proactive Parallel Decomposition

Dynamic decomposition is based on an attempt to minimize processor interruption. It can, however, in the worst case postpone re-balancing the workload among processors until a parallel computation is reduced to a sequential process. Let us now consider the effect of looking ahead, being as proactive as is reasonably possible. To accomplish this, we maintain a job queue and so that unexplored subtrees can be factored out, stored temporarily, and fed to processors when they are needed to keep every machine continuously busy. At the same time, we seek to eliminate the use of the NFS file server and file I/O by utilizing sockets for inter-processor communication. This approach opens socket connections between a driver and multiple clients, as illustrated in Figure 5.

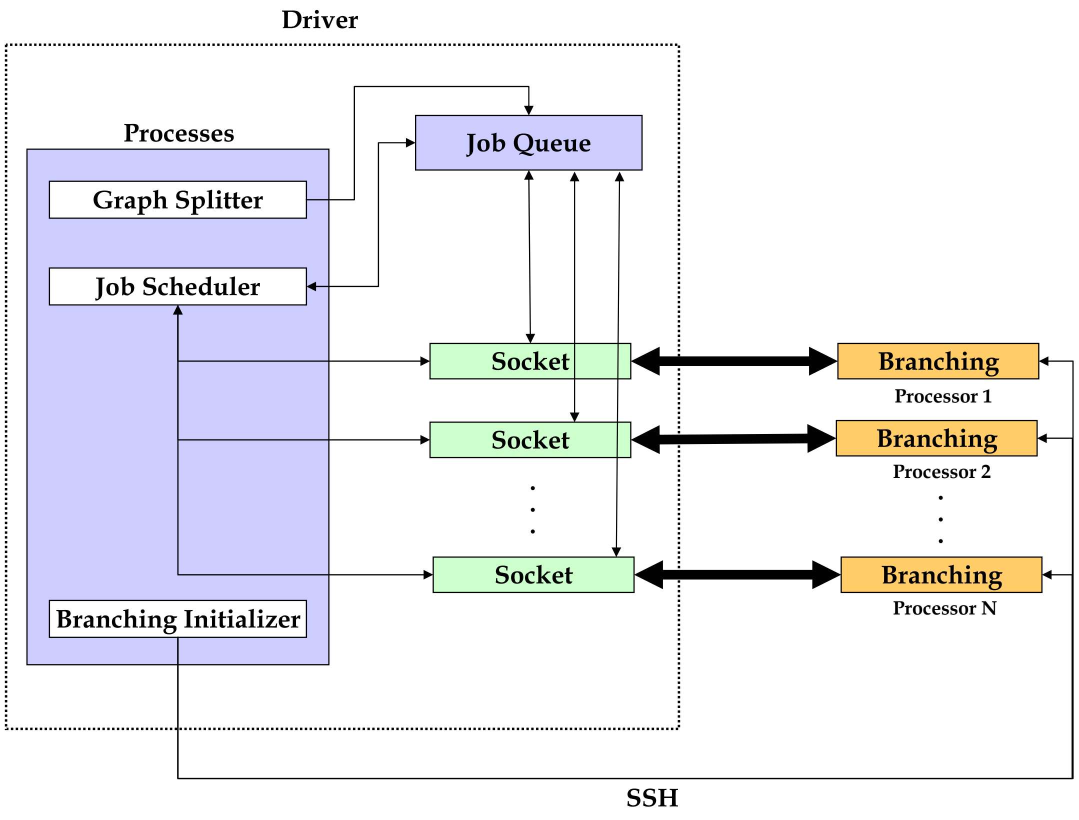

Figure 5: A proactive parallel decomposition strategy.

The central program (driver) maintains a job queue, initially containing the subtasks (subgraphs) resulting from the decomposition of the original problem instance after kernelization. Once initialization is complete, workload splitting and re-balancing is performed in a distributed fashion. The processors themselves, not some central controller, seize new subtasks and fork off others. It is easy to decide what to do whenever a processor becomes free. Said processor simply seizes the next job, if there are any, from the queue. The difficult question is in deciding when a job should be halted and its work divided up to ensure that the queue is not starved. The overhead required to split a processor's subgraph should not outweigh the computation time that can be saved across all machines. To try to ensure this, we split only "large subgraphs," that is, those that appear to require a considerable amount of computation. We also seek to avoid re-traversing any part of the search space. This is addressed by splitting a subtree at its current computational location. See Figure 6.

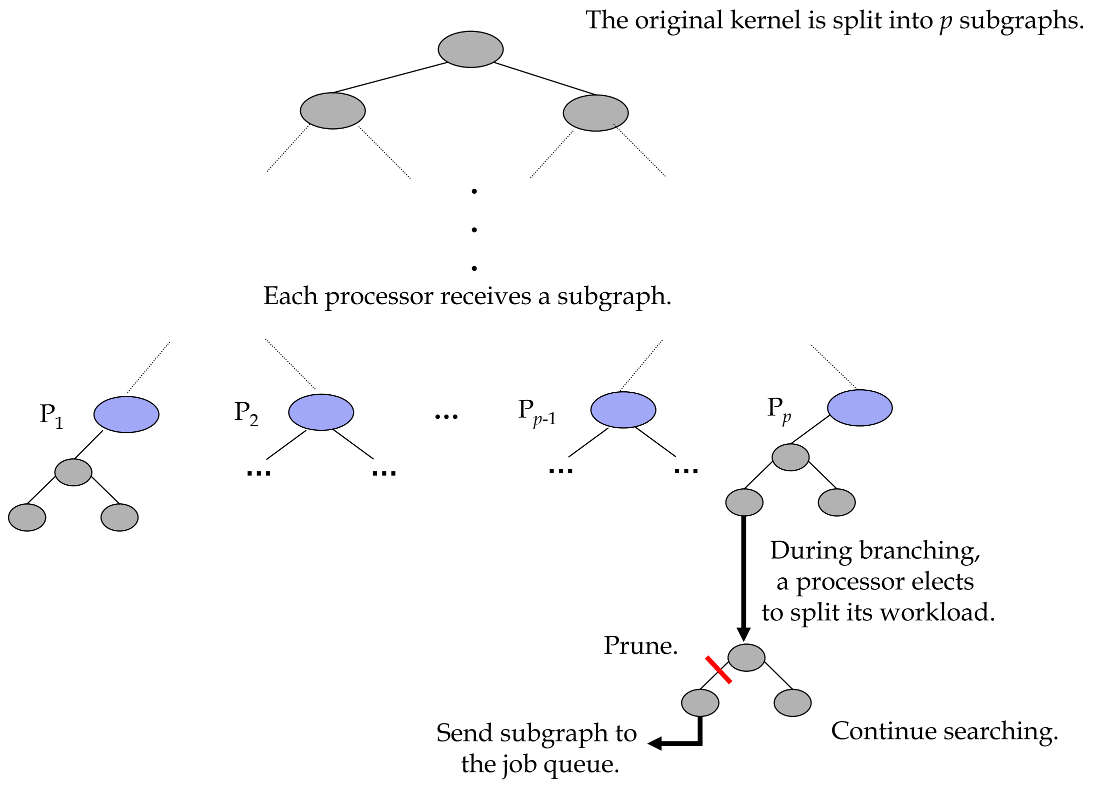
The original kernel is split into p subgraphs.
Each processor receives a subgraph.
P_1
P_2
$P_{p\text{-}1}$
P_p
During branching, a processor elects to split its workload.
Prune.
Send subgraph to the job queue.
Continue searching.

Figure 6: Distributed subtree splitting.

We first proposed this general sort of approach in [5]. We have worked on it since that time, and now have a complete design, a working implementation, and extensive empirical results. Representative comparisons of dynamic versus proactive decompositions are listed in Table 4. These experiments were run on protein domain data using eight 500MHz processors. As is evident from these numbers, proactive decomposition, despite its additional overhead, can perform significantly better than merely waiting to redistribute the workload until only one processor is active.

Graph		Cover	Instance	Parallel Decomposition	
Name	Size	Size	Type	Dynamic	Proactive
globin-3	972	165	yes	41 sec	23 sec
globin-3	972	164	no	43 sec	29 sec
globin-7	972	350	yes	1 min 30 sec	47 sec
globin-7	972	349	no	24 min 34 sec	17 min 49 sec
globin-9	972	378	yes	4 min 5 sec	3 min 47 sec
globin-9	972	377	no	20 min 41 sec	11 min 49 sec
sh2-3	839	246	yes	43 sec	22 sec
sh2-3	839	245	no	43 sec	27 sec

Table 4: A comparison of dynamic versus proactive parallel decomposition times.

In collaboration with our colleagues in neuroscience, we also ran comparisons using enormous sets of *mus musculus* microarray data. Each set represented 12,422 genes (vertices) and millions of interactions (edges). Depending to some extent on the particular value of k employed, kernelization generally cut the size of the graph by about half or so. On the reduced graph we used 32 processors, each running at 500 MHz. See Table 5.

Graph		Cover	Instance	Parallel Decomposition		
Name	Size	Size	Type	Static	Dynamic	Proactive
mus74-50	12,422	12,053	yes	6 days+	2 hr 51 min	2 hr 8 min
mus74-50	12,422	12,052	no	6 days+	5 hr 16 min	3 hr 57 min

Table 5: Additional comparisons of static, dynamic and proactive decomposition times.

It's probably safe to say that problems as large as those listed in Table 5 were until recently considered by most researchers to be hopelessly out of reach. Even when you consider that a binary search was performed to find the maximum size clique, and that therefore our routines were called multiple times, this entire operation took us just slightly more than a day to solve with current methods. Yet solving it at all was probably unthinkable just a short time ago. With the number of genes (vertices) set at n=12,422 and the optimal clique size turning out to be k=(12,422-12,053)=369, just imagine a straightforward $O(n^k)$ clique-find algorithm on a problem of that size!

12 Conclusions and Directions for Future Research

Coupling FPT-inspired algorithms with advanced computing platforms appears to be a fruitful approach to the development of scalable parallel algorithms for difficult optimization problems. Certain features, load balancing for instance, are critical. Some of our methods have already been incorporated into ClustalXP [2], the parallel, high-performance release of Clustal. ClustalXP currently runs on a 96-processor Beowulf cluster.

The payoff in collaboratory efforts can be huge. For example, based on the cliques we derived from microarray data, neurobiologists have identified what appear to be both network structures and gene roles in intra-cellular transport that were previously unrecognized. Because biological datasets and their computational demands can be so enormous, we are currently porting our codes to supercomputers at Oak Ridge National Laboratory. Especially attractive are a 512-processor SGI Altix and a 256-processor Cray X1. The former is particularly well-suite for our purposes because, in addition to raw speed and massive parallelism, it has a whopping two terabytes of shared memory available. The latter is also very appealing due to its superior speed and special built-in features for vector operations, which our codes use for graph manipulation and accounting.

We are also currently incorporating hardware acceleration into our on-campus implementations, in an effort to handle particularly recalcitrant subproblems. Through an infrastructure grant from NSF [13], we have brought on line a research cluster containing a dozen high-performance Unix boxes and eight PCs augmented with FPGA boards. We are prototyping and testing VHDL versions of our codes now. Reconfigurable technology is particularly well-suited to applications in which computation is in great demand while I/O requirements are light. Thus branching, but probably not kernelization, is an ideal candidate for this opportunity. Thus far our hardware accelerated branching cores are able to handle subgraphs with up to 256 vertices, and appear to show speedups in the 100-150 range.

Thus far, we have not employed a standard message-passing library, such as MPI [15, 21] or PVM [20]. This is because active processes have no need to communicate with one another. Moreover, our intent is to focus on scalability, so that we can employ truly heterogeneous computational resources with an arbitrary number of processors. Nevertheless we are open to this approach in future work. It might be useful in load-balancing, for example, if one wishes to look at the number and types of processors available before deciding when to split off a subtask and send it to the job queue.

Acknowledgments

We are grateful to Mike Fellows and Fran Rosamond for their collaboration and encouragement, to Bill Cook for providing us with the streamlined linear programming routines we employed in our reduction algorithms, and to Frank Dehne for access to some of the data sets we used in preliminary testing of our implementations.

References

[1] F. N. Abu-Khzam. *Topics in Graph Algorithms: Structural Results and Algorithmic Techniques, with Applications*. PhD thesis, Department of Computer Science, University of Tennessee, 2003.

[2] F. N. Abu-Khzam, J. Cheetham, F. Dehne, M. A. Langston, S. Pitre, A. Rau-Chaplin, P. Shanbhag, and P. J. Taillon. *ClustalXP*. See `http://ClustalXP.cgmlab.org/`.

[3] F. N. Abu-Khzam, R. L. Collins, M. R. Fellows, M. A. Langston, W. H. Suters, and C. T. Symons. Kernelization algorithms for the vertex cover problem: Theory and experiments. In *Proceedings, Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2004. To appear.

[4] S. Agrawal, J. Dongarra, K. Seymour, and S. Vadhiyar. *NetSolve: Past, Present, and Future - A Look at a Grid Enabled Server*. Wiley Publishing Company, 2003.

[5] N. E. Baldwin, R. L. Collins, M. A. Langston, M. R. Leuze, C. T. Symons, and B. H. Voy. High performance computational tools for motif discovery. In *Proceedings, IEEE International Workshop on High Performance Computational Biology (HiCOMB)*, 2004. To appear.

[6] J. F. Buss and J. Goldsmith. Nondeterminism within $\mathcal{P}$. *SIAM Journal on Computing*, 22:560–572, 1993.

[7] K. Cattell and M. J. Dinneen. A characterization of graphs with vertex cover up to five. In *Proceedings, International Workshop on Orders, Algorithms and Applications*, pages 86–99, 1994.

[8] J. Cheetham, F. Dehne, A. Rau-Chaplin, U. Stege, and P. J. Taillon. Solving large FPT problems on coarse grained parallel machines. *Journal of Computer and System Sciences*, 67:691–706, 2003.

[9] J. Chen. Parameterized computation and complexity: A new approach dealing with $\mathcal{NP}$-hardness, 2004. Preprint.

[10] J. Chen, I. Kanj, and W. Jia. Vertex cover: further observations and further improvements. *Journal of Algorithms*, 41:280–301, 2001.

[11] W. Cook. Private communication, 2003.

[12] F. Dehne. Private communication, 2003.

[13] J. Dongarra, M. W. Berry, M. Beck, J. Gregor, M. A. Langston, T. Moore, J. S. Plank, P. Raghavan, M. G. Thomason, R. C. Ward, and R. M. Wolski. *SInRG, A Scalable Intracampus Research Grid*. See `http://www.cs.utk.edu/sinrg`.

[14] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer-Verlag, 1999.

[15] J. Dongarra et al. MPI: A message-passing interface standard. *International Journal of Supercomputing Applications*, 8(3/4), 1994.

[16] M. R. Fellows. Parameterized complexity: The main ideas and some research frontiers. *Lecture Notes in Computer Science*, 2223:291–307, 2001.

[17] M. R. Fellows and M. A. Langston. Nonconstructive advances in polynomial-time complexity. *Information Processing Letters*, 26:157–162, 1987.

[18] M. R. Fellows and M. A. Langston. Fast search algorithms for layout permutation problems. *International Journal of Computer Aided VLSI Design*, 3:325–342, 1991.

[19] M. R. Fellows and M. A. Langston. On search, decision and the efficiency of polynomial-time algorithms. *Journal of Computer and Systems Sciences*, 49:769–779, 1994.

[20] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine–A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, Mass., 1994.

[21] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994.

[22] D. Hochbaum. *Approximation Algorithms for $\mathcal{NP}$-hard Problems*. PWS, 1997.

[23] S. Khuller. The vertex cover problem. *ACM SIGACT News*, 33:31–33, June 2002.

[24] U.S. National Institutes of Health National Library of Medicine. *PubMed Central*. See `http://www.pubmedcentral.nih.gov/`.

[25] G.L. Nemhauser and L. E. Trotter. Vertex packings: Structural properties and algorithms. *Mathematical Programming*, 8:232–248, 1975.

[26] R. Niedermeier and P. Rossmanith. A general method to speed up fixed-parameter tractable algorithms. *Information Processing Letters*, 73:125–129, 2000.

[27] K. Park and H. Lee. On the effectiveness of route-based packet filtering for distributed dos attack prevention in power-law internets. In *Proceedings, ACM Conference of the Special Interest Group on Data Communication*, pages 15–26, 2001.

Appendix: The NetSolve Problem Description File

The following problem description file is used to describe *pbranch*, a parallel branching program, to the NetSolve middleware system. With this information, pbranch is made visible and available to all users within the NetSolve community. This data is also used to establish the appropriate programming interface for remote procedure calls.

```
@PROBLEM pbranch /* Problem name visible to NetSolve clients */
@FUNCTION pbranch /* Function name that will be invoked to solve the problem */
@LIB -L$(LIBLANGSTONDIR) -llangston /* Library link */
@LANGUAGE C /* Language of the underlying library */
@MAJOR ROW /* Specifies row major order, since the library language is C */
@PATH langston /* Path-like naming convention */
@NON_MOVEABLE 1 /* Problem is not to be moved to another server */
@DESCRIPTION /* Begin text description */
Remotely executes pbranch code.

@INPUT 3 /* Number of input objects for the problem */
@OBJECT FILE infile /* Object type and name, to be followed by description */
Parameters file
@OBJECT FILE matfile /* Object type and name, to be followed by description */
Matrix file
@OBJECT SCALAR I id /* Object type and name, to be followed by description */
Processor id

@OUTPUT 2 /* Number of output objects for the problem */
@OBJECT FILE outfile /* Object type and name, to be followed by description */
Informational file
@OBJECT FILE coverfile /* Object type and name, to be followed by description */
Cover file

@CALLINGSEQUENCE /* Argument order for the programming interface */
@ARG I0
@ARG I1
@ARG I2
@ARG O0
@ARG O1

@CODE /* Begin psuedo-code section */
extern void pbranchexec(char* infile, char* matfile, int id,
                              char* outfile, char* coverfile);
pbranchexec(@I0@,@I1@,*@I2@,@O0@,@O1@); /* Initiate pbranch on remote machine */
@END_CODE /* End psuedo-code section */
```