Parallel Vertex Cover: A Case Study in Dynamic Load Balancing

Dinesh P. Weerapurage, John D. Eblen, Gary L. Rogers Jr. and Michael A. Langston

Department of Electrical Engineering and Computer Science
University of Tennessee
Knoxville TN 37996-3450
USA
{dpremala, eblen, grogers, langston}@eecs.utk.edu

Abstract

The significance of dynamic parallel load balancing is considered in the context of fixed parameter tractability. The well known vertex cover problem is used as a case study. Several algorithms are developed and tested on graphs derived from real biological data. Implementations are carried out on the Kraken supercomputer, currently the world's fastest computational platform for open science. We show that for certain difficult instances of biological data graphs our approach scales well up to 2400 processors.

Keywords: parallel computation, load balancing, data mining, genome scale analysis

1 Introduction

Given a graph $G = < V, E >$ and an integer k ($k \leq n$, $n = |V|$), the vertex cover problem asks whether V contains a subset C of size at most k so that every member of E has at least one endpoint in C. Applications are myriad, in large part because vertex cover is so easily transformed into clique and its dual (independent set). A huge number of domains are amenable (Bomze et al. 1999). Specific examples include bio-informatics (Samudrala 2006), chemistry (Rhodes et al. 2003), electrical engineering (Prihar 1956), and even social networks (Luce & Perry 1949). Vertex cover is fixed parameter tractable (FPT) for every fixed k (Downey & Fellows 1999). The asymptotically fastest currently known algorithm runs in $O(1.2738^k + kn)$ time (Chen et al. 2006).

Despite the potential efficiencies offered by FPT, vertex cover remains $\mathcal{NP}$-complete (Garey & Johnson 1990). Accordingly, computational burdens are often onerous. Here we are interested chiefly in effective parallel load balancing schemes. We shall describe our work as follows. In Section 2, we present a simple sequential FPT vertex cover algorithm. Next, in Section 3, we devise a relatively straightforward parallel method employing a static form of load balancing to distribute the effort across processors. In Section 4, we develop a more complex but dynamic load balancing approach. A variety of experimental results are discussed in Section 5. In a final section we list a number of conclusions and directions for future research.

This research has been funded by the U.S. Department of Energy under the EPSCoR Laboratory Partnership Program. It has also been supported by an allocation of advanced computing resources provided by the U.S. National Science Foundation. Computations were performed on Kraken, a Cray XT5 housed at the National Institute for Computational Sciences, Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA.

2 A Sequential Vertex Cover Algorithm

In the decision version of vertex cover, we require only a "yes" or "no" answer. In the case of "yes" we also solve the search version by producing a satisfying cover. To solve the optimization version, we employ the decision algorithm along with binary search to find the smallest k for which the answer is "yes."

2.1 Kernelization

FPT's success as an algorithm design paradigm relies largely on kernelization, a form of problem reduction that ensures a compute core whose size depends only on k and not on n ($n = |V|$). If (G, k) is an instance of vertex cover, an effective kernelization routine produces another instance (G', k'), in which $k' \leq k$ and the number of vertices in G' is bounded above by some polynomial function of k'. G has a cover of size k if and only if G' has a cover of size k'. Numerous kernelization procedures have been proposed (Abu-Khzam et al. 2001, Chen et al. 2001). Those with the most theoretical appeal tend to be the most challenging to implement, while those with the simplest structure are often the most effective on real data. We have thus adopted only the following kernelization rules for this study.

Rule 1 (The degree one rule):

A vertex with degree one is excluded from the cover. This is because there is no gain in including a pendant vertex to cover its only neighbor.

Rule 2 (The high degree rule):

A vertex whose degree exceeds k must be in the cover. Otherwise, all of its neighbors must be in the cover, which is impossible.

In addition to the above two rules, there are kernelization algorithms to preprocess 2-degree vertices, 3-degree vertices and even higher degree vertices, but they become successively more difficult to implement. Other kernelization methods include linear programming algorithms (Abu-Khzam et al. 2006) and crown decomposition (Abu-Khzam et al. 2001). Analyzing the performance of different kernelization algorithms and implementations is a broader topic that requires a separate paper. Here, we focus our efforts on the hard computational core that kernelization produces, because that is where exponential run time occurs.

2.2 Decomposition and search

After reducing the graph using the two kernelization rules just described, the kernel should be searched for solutions. In FPT terminology this step is known as "branching." The challenge is that the reduced problem core still might have an exponential number of solution candidates. We use a branching algorithm, which utilizes a search tree, to traverse through the search space. Each branch of the search tree represents a possible solution, and after searching the tree to a certain depth, we can make a decision on whether that particular branch contains a solution or not. This is an exhaustive search, and sometimes the entire solution space must be searched, such as when no solution exists.

In the branching tree, each branch is a possible choice for building the vertex cover. Each branch adds one or more vertices to the cover, and k is updated accordingly. If the value of k becomes zero or less than zero, a decision is made about the cover. If it is a valid cover, the search is ceased. Otherwise, searching is resumed on another branch.

Branching is illustrated in Figure 1. Letters in the box represent the current vertex cover while the graphs on either side show the modified graph. Black vertices represent vertices that have been deleted, whereas white vertices are vertices still in the graph. The neighbors of vertex v are shown as $N(v)$ and number of neighbors as $|N(v)|$. First, vertex v is selected for the vertex cover in the left branch, which can be seen in the left box. Then the k value is reduced by one. The right branch contains a choice without vertex v. All edges connected to v should be covered. Therefore all $N(v)$ should be in the vertex cover. Thus, the right box has vertex cover without v but including $N(v)$. Also, k is updated to $k - |N(v)|$.

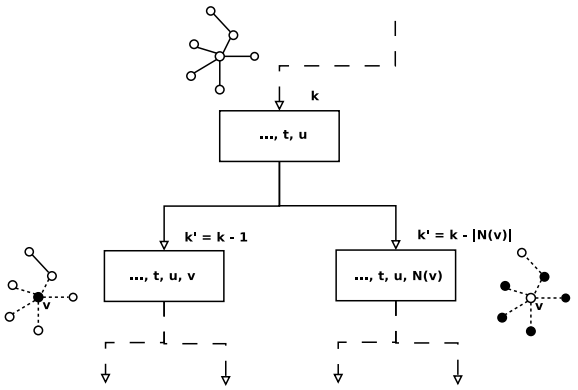

Figure 1: A branching algorithm

Figure 2 shows a simple graph with 7 vertices and 9 edges and Figure 3 shows the decomposition steps for $k = 4$ on this graph. As in Figure 1, numbers in the boxes show the partial cover found at each branch. Graphs on either side illustrate the modified graph for each branch. Dotted lines are deleted edges, black vertices are deleted vertices, and white vertices are vertices in the graph. The k value in the upper right hand corner of each box is the updated k' value for the modified graph G'. When the cover size is k and all edges in the graph are covered, it is a valid vertex cover, depicted as a box within a box. Otherwise it is an invalid cover, depicted as a dashed-line box within a box. In this example, a current highest degree vertex is selected for the vertex cover. A vertex cover is found in the leftmost branch, and the search can be ceased. In the case where a graph might be structured in such a way that covers are in the rightmost branches, the algorithm has to search through all left branches before reaching the right side. It is important to note that kernelization can also be applied during branching in subsequent recursive calls. This is termed "interleaving" (Niedermeier & Rossmanith 2000), which can help to reduce problem instances and improve algorithm performance.

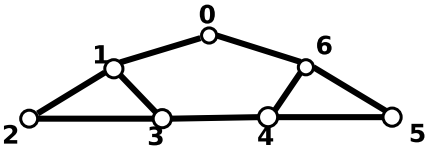

Figure 2: An example graph

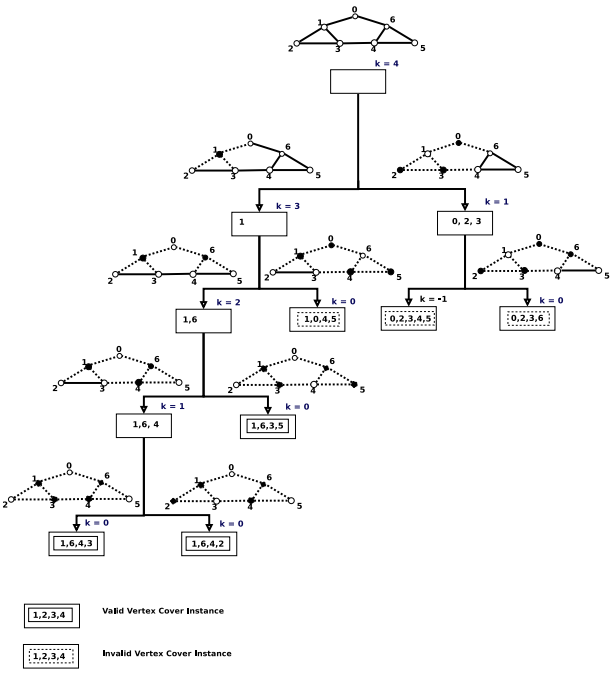

Figure 3: Branching tree for graph of Figure 2, with k = 4

3 Parallel Vertex Cover

Decomposing a graph is a computationally demanding operation, and thus distributing work among several processors will help to decompose and search the graph more effectively. During the decomposition stage, branches are independent of each other, therefore branches can be searched separately and results are reported back when the computation is completed. The implementation has been done using MPI (Dongarra & Walker 1994), (Gropp et al. 1999) given that it is widely adopted and scales well. We employ a master–worker architecture, where each processor is assigned an independent branch to compute.

In Figure 4, experimental data on real biological data graphs are shown. The data sets, related to "Folic acid deficiency effect on colon cancer cells," "Low concentrations of 17beta-estradiol effect on breast cancer cell line," and "Interferon receptor deficient lymph node B cell response to influenza infection," were obtained from NCBI (NCBI 2010). The graphs were created using genes as vertices and Pearson correlation p-values as the weights of edges between all pairs of genes. The graphs were then thresholded using p-values (0.45, 0.40, 0.35 and 0.30) and converted to undirected graphs. This is done by removing edges between vertex pairs that have a p-value greater than the threshold value. The minimum vertex cover of each graph was found using both the sequential algorithm

Graph	Sequential	Parallel
fo-0.45	2384.67	2400.00
fo-0.40	10023.07	9870.77
fo-0.35	29792.09	28502.76
fo-0.30	>1day	>1day
est-0.45	915.18	894.21
est0.40	14004.66	2630.92
est-0.35	12289.54	12043.73
est-0.30	56335.00	56150.65
inf-0.45	305.34	290.04
inf-0.40	696.76	671.87
inf-0.35	2089.13	2005.68
inf-0.30	7440.08	7243.81

Table 1: Sequential and parallel execution times (in seconds) for sample application graphs

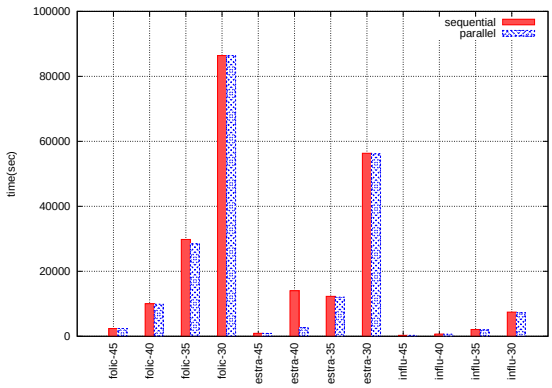

Figure 4: A comparison of sequential and parallel performance

and the parallel algorithm using a computing cluster that consisted of 32 Intel 3.2GHz processors, each with 4GB of main memory. Runs were limited to 24 hours (86400 seconds). Table 1 summarizes the results. (The folic-30 graph did not finish within 24 hours and the program was terminated.)

When using n processors for parallel algorithms, n-times speedup is expected, but these experiments fall far short of that goal. For instance, with the sequential algorithm folic-35 took 29792 seconds. With 32 processors, it took 28502.76 seconds. So even though 96% improvement in execution time was hoped, the actual execution time speedup was as low as 4.3%. The process utilization for the static load balancing algorithm for folic-35 is shown in figure 5. Low throughput happens as a result of uneven process utilization. (One processor does nearly all of the work.)

As we have seen in Figure 4, assigning independent branches to different processors is not always optimal, in terms of runtime. Given that not all branches are equal in terms of complexity, some processors work longer on difficult branches and other processors finish their job quickly. Even though searching is distributed to different processors, the workload will reduce to only a few number of busy processors quickly. If we can use idling processors to assist difficult branches it will help to increase overall efficiency. A load balancing algorithm will help to distribute the load across all processors evenly.

4 Dynamic Load Balancing

We applied a scheduler-based load balancing algorithm to the parallel vertex cover algorithm. Several load balancing mechanisms for parallel algorithms have been in-

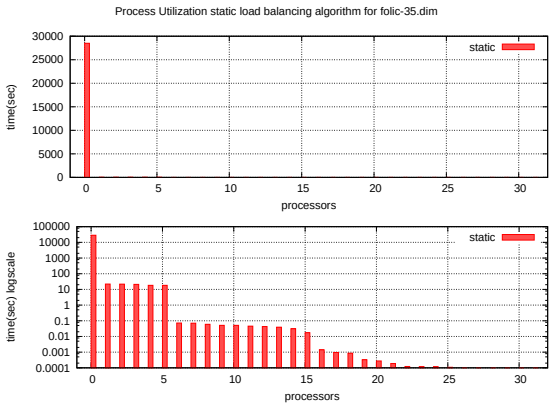

Figure 5: Processor utilization in static load balancing

vestigated in (Kumar et al. 1994). Six receiver initiated and three sender initiated algorithms have been discussed. Among them, a scheduler based algorithm (receiver initiated), which was also proposed in (Patil & Banerjee 1989), is more suitable for load balancing the parallel vertex cover algorithm. The sender initiated algorithm would be a push model that is hard to implement as the working processors finish time is unknown during execution. Also, since our algorithms select the current highest degree vertex during branching, we suspect that the leftmost branch is the hardest branch. So for balanced process utilization, idling processors should assist the leftmost branch.

Previously, at least two dynamic load balancing algorithms for parallel vertex cover have been suggested in (Abu-Khzam et al. 2006) and (Baldwin et al. 2004). Also, load balancing has been discussed in (Taillon 2007). Stack splitting and search frontier splitting methods, which were introduced in (Reinefeld & Schnecke 1994) and (Reinefeld 1994) have been discussed. The algorithm discussed in this paper is slightly different than the above two algorithms because this algorithm is tightly coupled with the branching tree whereas the aforementioned algorithms are more general load balancing algorithms. However, one could define it as a variation of the stack splitting method (Reinefeld 1994), because work is partitioned on demand to be delivered to the requester. The hardest instance is designated as the donor, and it is the job donating process. Therefore, the scheduler does not need to maintain a job queue. As we explained in the previous paragraph the hardest instance occurs in the leftmost branch. Processors that need more jobs contact the scheduler, who redirects job requests to the donor. Jobs are then delivered to the requesting processor directly. This process is shown in the diagram in Figure 6.

The diagram in Figure 7 shows the termination process. The scheduler is notified when the donor runs out of jobs. The scheduler then terminates the donor, and any processor requesting more work will be issued a termination signal.

Execution times in Figure 4 are compared with execution times for the dynamic load balancing algorithm in Figure 8. Some experiments with dynamic load balancing finished so quickly that DLB bars are not visible for some graphs. In order to have a clearer comparison, Figure 9 uses a log scale for execution times.

Although the parallel vertex cover algorithm with dynamic load balancing outruns both the parallel vertex cover algorithm without load balancing and the sequential algorithm, a load imbalance still occurs once the donor finishes to address this issue donor update algorithm is implemented.

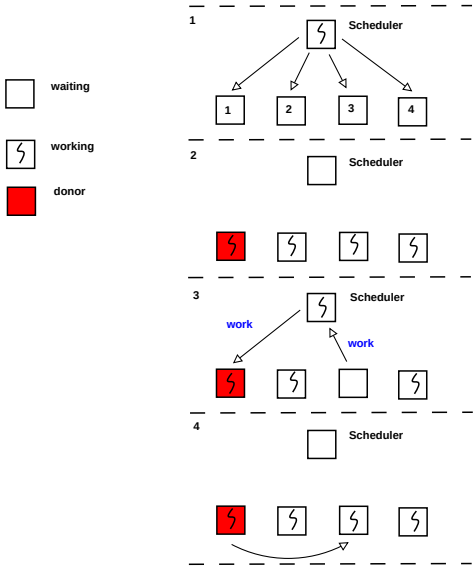

Figure 6: A dynamic load balancing algorithm

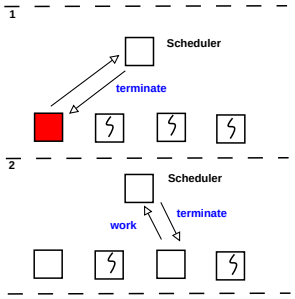

Figure 7: Load balancing termination

Graph	Sequential	Parallel	DLB
fo-0.45	2384.67	2400.00	108.28
fo-0.40	10023.07	9870.77	430.19
fo-0.35	29792.09	28502.76	1265.137
fo-0.30	>1day	>1day	6448.77
est-0.45	915.18	894.21	43.74
est0.40	14004.66	2630.92	121.90
est-0.35	12289.54	12043.73	525.93
est-0.30	56335.00	56150.65	2485.52
inf-0.45	305.34	290.04	17.35
inf-0.40	696.76	671.87	33.70
inf-0.35	2089.13	2005.68	96.00
inf-0.30	7440.08	7243.81	345.50

Table 2: Sequential, parallel and DLB execution times (in seconds)

4.1 Donor Update Algorithm

For the previous algorithm, after the initial job distribution, one processor will be designated as the donor, which is terminated when the work in the assigned branch is finished. All processors with subsequent calls for more jobs will be terminated too. At this moment any processor that has a difficult branch will work longer while others are being terminated. In order to achieve balanced process utilization, processors that finished should be used to support those still working.

To solve this problem, we create a donor update algorithm, which works as follows. The initial donor keeps track of the degree structures of the jobs that were distributed to workers. Upon termination the donor calculates the average degree for the last jobs distributed to the workers. Then these average degrees are sent to the scheduler, who selects the next donor. The scheduler selects the worker with the highest average degree job as the new donor.

It is important to note that only the first donor keeps track of the jobs sent to workers, because keeping track of jobs is expensive both in terms of memory and cycles. Moreover, initial donor is the hardest instance and all other processors have work that is donated from the initial donor. Therefore we assume that the jobs later donors have are not as hard as the jobs initial donor had. It is not effective to spend resources to obtain information from relatively easy jobs.

Initial donor sends degree information to the scheduler upon termination whereas all other donors terminate without sending additional information. The scheduler uses degree information to select next donors. After the initial donor exited, degree information for workers who ask for more jobs will be removed. As a result, degree information for the busy workers who later will be selected as donors will be remained. After all degree information are used, workers are issued the termination signal. The donor update process is depicted in Figure 10.

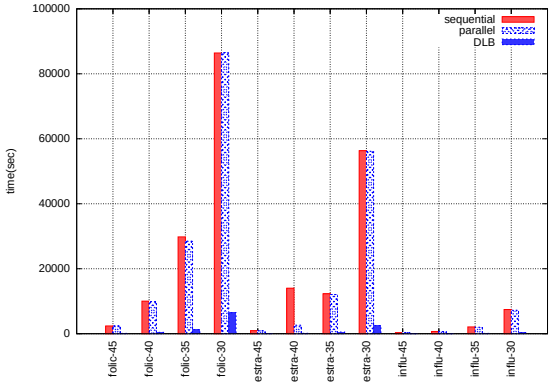

Figure 8: A comparison of sequential, parallel and DLB performance

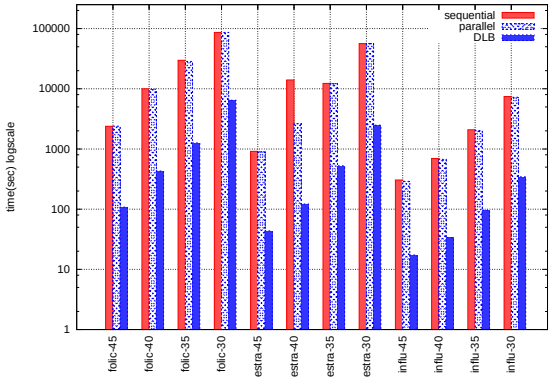

Figure 9: A log scale view of relative performance

5 Experimental Results and Discussion

Twelve biological data graphs were tested using the decision version of the vertex cover algorithm, and a cluster of 32 Intel 3.2GHz processors with 4GB of main memory

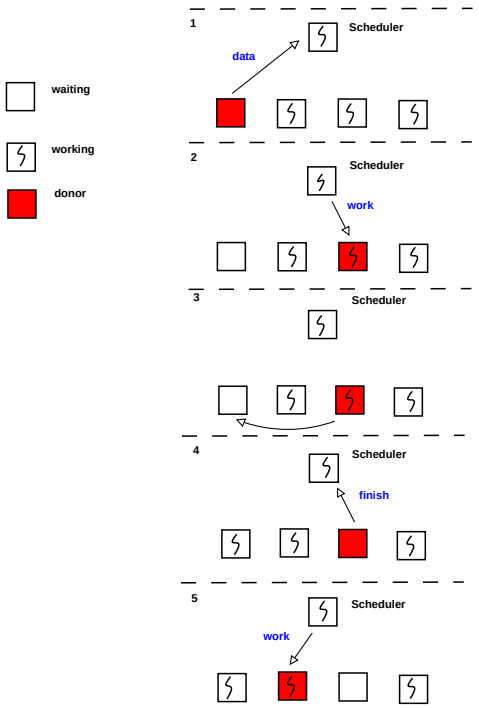

Figure 10: Donor update algorithm

Graph	Sequential	Parallel	DLB
fo-0.45	306.52	293.70	9.18
fo-0.40	147.13	140.30	35.18
fo-0.35	647.76	646.61	42.84
fo-0.30	6837.19	6831.89	339.28
est-0.45	23.46	23.43	2.5
est-0.40	3728.79	3637.11	33.00
est-0.35	836.74	837.04	55.13
est-0.30	3831.91	3812.89	216.68
inf-0.45	1.0	-	-
inf-0.40	0.6	-	-
inf-0.35	789.62	757.09	45.13
inf-0.30	3.0	3.3	0.59

Table 3: Times for hardest "yes" instances

was used for testing. Results in Table 3 are for the hardest "yes" instance while data in Table 2 are for the hardest "no" instance. When the parallel program is working on a "yes" instance, the search will be terminated as soon as one processor finds a solution, whereas in a "no" instance, the whole search space has to be searched. For example, in Table 2 for influenza-30, the graph vertex cover program spent 7440 seconds on the hardest "no" instance. On the other hand, in Table 3, the program spent only 3.0 seconds to find a "yes" instance. Thereby, the execution time of a "yes" instance cannot be used for meaningful comparisons. Hence execution times for "no" instances were used. A sequential program for a "yes" instance for influenza-0.45 and influenza-0.40 graphs finished very quickly (less than 1.0 sec). Thus a parallel program was not required.

Execution times for sequential algorithm, static load balancing (or parallel algorithm without explicit load balancing algorithm) and dynamic load balancing algorithm were compared in Table 2. Although the static load balancing algorithm uses 32 processors, it did not gain expected performance. As we saw in Figure 5, that happened as a result of uneven distribution of work load. The parallel algorithm with static load balancing will boil down to a small number of busy processors quickly, therefore producing little improvement in execution time. In order to address this issue, a dynamic load balancing algorithm was introduced. The algorithm is explained in detail in Section 4. Dynamic load balancing gives 20–25 times speedup for the biological data graphs that have been selected. However, this speedup is still sub-linear. The main reason for this slowdown is communication between donor and workers, which is indeed inevitable. The intent of this dynamic load balancing algorithm is to have regular load distribution among the workers and to minimize idle time.

Table 2 lists execution time for the sequential, parallel and dynamic load balancing algorithms. Based on the results in Table 2, folic-30, estradio-30 and influenza-30 are the hardest instances in their category. Hardest instances were selected for further analysis of process utilization by the dynamic load balancing algorithm and the donor update algorithm. In this experiment, processor idling time is considered as the time the processor is idle. Time for communication is not counted as idling time.

Graph	Execution Time	Idling Time
fo-0.30	731.90	9.15
est-0.30	317.50	5.24
inf-0.30	32.36	0.92

Table 4: Execution and idling times for DLB on 120 processors

Graph	Execution Time	Idling Time
fo-0.30	694.16	62.87
est-0.30	276.00	25.82
inf-0.30	34.71	3.88

Table 5: Execution and idling times for DU on 120 processors

After completing all tests using the aforementioned cluster, load balancing codes were moved to the highly capable Kraken supercomputer at ORNL. Load balancing algorithms were tested for scalability ranging from 12 processors to 2400 processors. Each node on Kraken has two 2.6 GHz six-core AMD Opteron processors and contains 16 GB of main memory. Table 4 has information on execution time and idling time with the dynamic load balancing (DLB) algorithm. Also, Table 5 has execution time and idling time for the donor update (DU) algorithm. The DU algorithm has larger idling time compared to the DLB algorithm. Although DU algorithm has higher idling time it performs well compared to the DLB algorithm. Figure 11 has idling time as a percentage of execution time for folic-30 graph with 120 processors for both DLB and DU algorithms. DLB algorithm has less than 1% of idle time whereas DU algorithm has up to 10% of idle time. There is extra communication in DU compared to DLB as a result processors have to wait longer. However in DU collectively processors perform better than DLB because extra communication in DU make sure to occupy idling processors with work whereas in DLB after donor terminates workers who got harder job work harder while others terminating.

Both the DLB algorithm and the DU algorithm were tested for scalability using the hardest graph instances (folic-30, estradio-30 and influenza-30). The execution time plot for folic-30 is presented in Figures 12. Although execution times for folic-30, estradio-30 and influenza-30 were measured, only one plot of execution time is included because all three graphs are similar in shape. Normal measurement does not show any difference in execution time, so a log scale was applied. Note that the third plot from Figure 12 shows the difference in execution time between the DLB and DU algorithms more clearly than

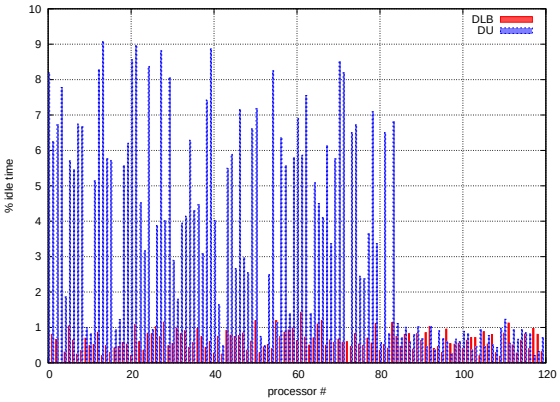

Figure 11: Percentage idle time for DLB and DU algorithms with folic-30 graph

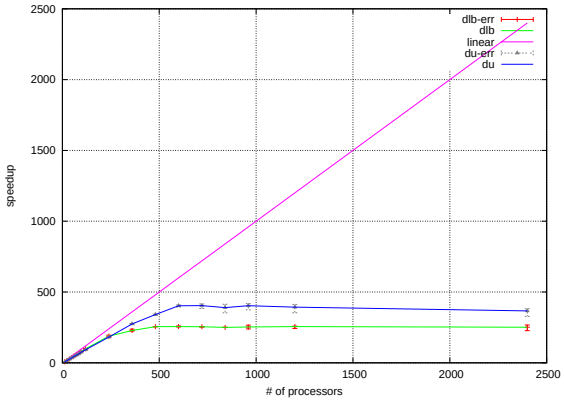

Figure 15: Speedups for influenza-30 graph on Kraken

the first plot. With smaller number of processors, both the DLB and the DU algorithm almost follow expected (linear) execution time, but for higher number of processors DU execution time is closer to the expected value than the DLB execution time. Note that speedup for each algorithm was computed by dividing sequential execution time by number of processors. Expected speedup is indicated as linear speedup on the plots. In addition an error bar representation is available on the speedup plots to show the consistency of the results.

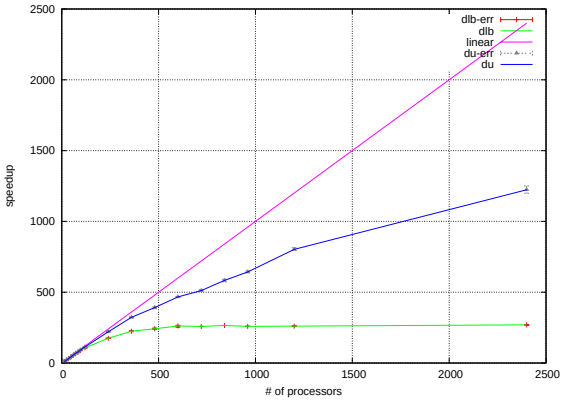

Figure 13: Speedups for folic-30 graph on Kraken

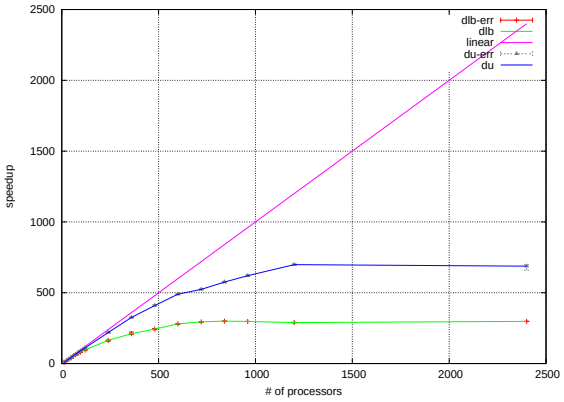

Figure 14: Speedups for estradio-30 graph on Kraken

Curves in Figures 13, 14 and 15 have noticeable bends because smaller samples were chosen for the experiment. The sample includes 12, 24, 36, 48, 72, 120, 240, 480, 600, 720, 840, 960, 1200 and 2400 processors. Plots were produced using data for the aforementioned sample. If experiments are done for a larger sample smoother curves can be produced.

The DU algorithm scales better for larger numbers of processors than the DLB algorithm. As seen in Figure 13, DLB does not scale up after 120 processors. Because after 120 processors donor finishes quickly and workers who get a harder job end up working longer, which leads to a smaller magnitude load imbalance again. On the other hand, the DU algorithm makes sure to assist harder jobs using idling processors until all harder jobs are finished. Therefore DU scales up well even up to 2400 processors.

Even though scalability of the DU algorithm can be clearly seen in Figure 13, in Figure 14 and in Figure 15 it is not as clear. Primarily this is because the estradio-30 and influenza-30 graphs are smaller compared to the folic-30 graph and thus not as hard. Since estradio-30 and influenza-30 graphs are not much harder there is not much work for larger number of processors hence communication overhead dominates the execution time.

After doing these experiments several times, maximum, average and minimum values for execution time are obtained. Average values use for plots, in addition minimum and maximum values are shown using an error bar representation. Error bars for all three graphs are small, which implies that execution time (and speedup) are consistent for both DLB and DU algorithms.

6 Conclusions and Direction for Future Research

Vertex cover remains an $\mathcal{NP}$-complete problem. As such, its computational requirements can be staggering. Yet it has many practical applications that demand exact solutions. Fixed parameter tractability is one approach for dealing with this conundrum. Kernelization and decomposition are key. Decomposition in particular can be based on independent branching actions, which are inherently parallel.

We initially developed a simple parallel decomposition algorithm. Different branches were statically assigned to different processors. Once a job finished, results were reported. Unfortunately, this scheme does not always scale well. Some branches are much harder than others, in which case one or only a few processors are left to do most of the work.

We therefore devised a dynamic load balancing algorithm. With it, the branch with what is perceived as the hardest task is selected as a donor, which then distributes

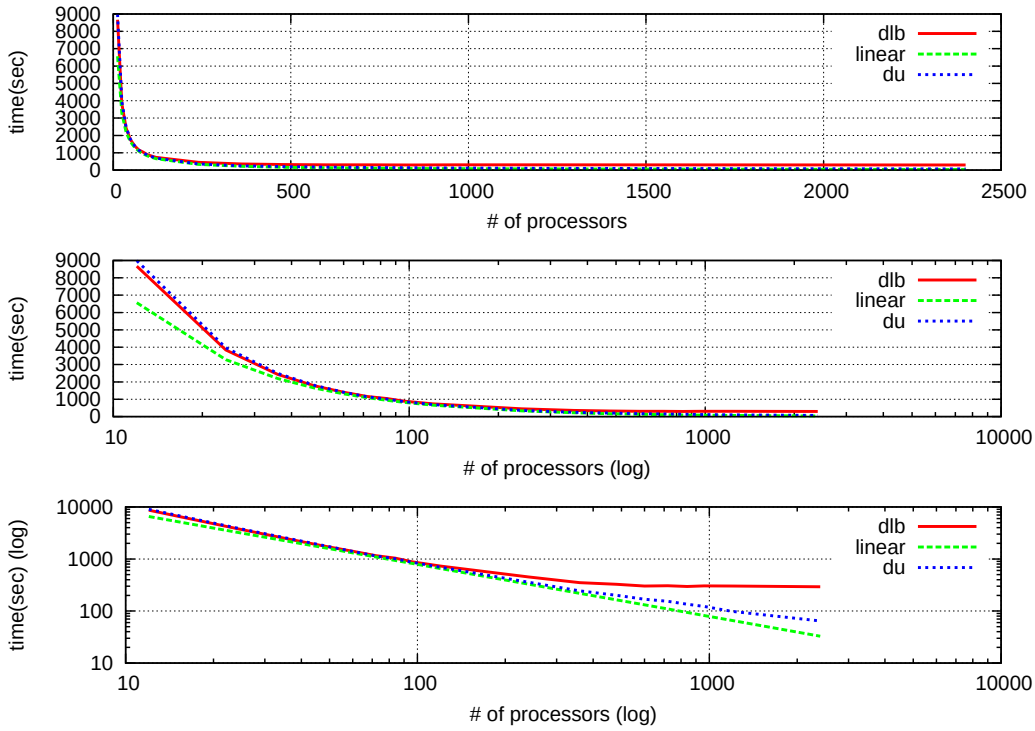

Figure 12: Execution times for folic-30 graph on Kraken

jobs to idling processors. Requests for more jobs will be made upon completion of a current job. There is a possibility of load imbalance, so a donor update algorithm is also implemented. When a current donor finishes, it will find a next donor and so on.

Primary testing was done using an in-house cluster containing 128 processors. More in-depth scalability analysis was performed on the Kraken supercomputer. With sufficiently difficult instances, dynamic load balancing shows linear speedup up to 2400 processors. On graphs without imposing execution times, we see only sub-linear speedup as communication overhead dominates.

We have previously observed numerous problems with static load balancing, curiously more so on real than on synthetic data. Thus this work was intended mainly as a case study. Nevertheless, we have found the switch to dynamic load balancing to be fairly straightforward and quite effective. We think it would be interesting to experiment with dynamic load balancing algorithms on other FPT problems. It may also be interesting to extend scalability testing to larger numbers of processors.

References

Abu-Khzam, F. N., Collins, R. L., Fellows, M. R., Langston, M. A., Suters, W. H. & Symons, C. T. (2001), 'Kernelization algorithms for the vertex cover problem: Theory and experiments (extended abstract)'.

Abu-Khzam, F. N., Langston, M. A., Shanbhag, P. & Symons, C. T. (2006), 'Scalable parallel algorithms for fpt problems', *Algorithmica* **45**(3), 269–284.

Baldwin, N., Collins, R. L., Langston, M. A., Symons, C. T., Leuze, M. R. & Voy, B. H. (2004), 'High performance computational tools for motif discovery', *Parallel and Distributed Processing Symposium, International* **10**, 192a.

Bomze, I. M., Budinich, M., Pardalos, P. M. & Pelillo, M. (1999), The maximum clique problem, *in* 'Handbook of Combinatorial Optimization', Kluwer Academic Publishers, pp. 1–74.

Chen, J., Kanj, I. A. & Jia, W. (2001), 'Vertex cover: Further observations and further improvements', *Journal of Algorithms* **41**, 313–324.

Chen, J., Kanj, I. A. & Xia, G. (2006), Improved parameterized upper bounds for vertex cover, *in* R. Krlovic & P. Urzyczyn, eds, 'Mathematical Foundations of Computer Science 2006', Vol. 4162 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 238–249.

Dongarra, J. J. & Walker, D. W. (1994), 'MPI: A message-passing interface standard', *Internaitonal Jornal of Supercomputing Applications* **8**(3/4), 159–416.

Downey, R. & Fellows, M. (1999), *Parameterized Complexity*, Springer , Berlin.

Garey, M. R. & Johnson, D. S. (1990), *Computers and Intractability; A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co., New York, NY, USA.

Gropp, W., Lusk, E. & Skjellum, A. (1999), *Using MPI (2nd ed.): portable parallel programming with the message-passing interface*, MIT Press, Cambridge, MA, USA.

Kumar, V., Grama, A. Y. & Vempaty, N. R. (1994), 'Scalable load balancing techniques for parallel computers', *J. Parallel Distrib. Comput.* **22**(1), 60–79.

Luce, R. & Perry, A. (1949), 'A method of matrix analysis of group structure', *Psychometrika* **14**, 95–116. 10.1007/BF02289146.
URL: *http://dx.doi.org/10.1007/BF02289146*

NCBI (2010), 'The national center for biotechnology information advances science and health by providing access to biomedical and genomic information.'. [Online;

accessed 02-May-2010].
URL: *http://www.ncbi.nlm.nih.gov/*

Niedermeier, R. & Rossmanith, P. (2000), 'A general method to speed up fixed-parameter-tractable algorithms', *Inf. Process. Lett.* **73**(3-4), 125–129.

Patil, S. & Banerjee, P. (1989), A parallel branch and bound algorithm for test generation, *in* 'DAC '89: Proceedings of the 26th ACM/IEEE Design Automation Conference', ACM, New York, NY, USA, pp. 339–343.

Prihar, Z. (1956), 'Topological properties of telecommunication networks', *Proceedings of the IRE* **44**(7), 927 –933.

Reinefeld, A. (1994), Scalability of massively parallel depth-first search, *in* 'In DIMACS Workshop', American Mathematical Society, pp. 305–322.

Reinefeld, A. & Schnecke, V. (1994), Work load balancing in highly parallel depth first search, *in* 'In Scalable High Performance Computing Conference', pp. 773–780.

Rhodes, N., Willett, P., Calvet, A., Dunbar, J. B. & Humblet, C. (2003), 'Clip: similarity searching of 3d databases using clique detection', *Journal of Chemical Information and Computer Sciences* **43**(2), 443–448. PMID: 12653507.
URL: *http://pubs.acs.org/doi/abs/10.1021/ci025605o*

Samudrala, Ram; Moult, J. (2006), 'A graph-theoretic algorithm for comparative modeling of protein structure', *Journal of Molecular Biology* **279**(1), 287–302.

Taillon, P. J. (2007), On improving fpt k-vertex cover, with applications to some combinatorial problems, PhD thesis, Carleton University, Ottawa, Canada.