A Decentralized Load Balancing Approach for Parallel Search-Tree Optimization

F. N. Abu-Khzam* and A. E. Mouawad†

*Department of Computer Science and Mathematics
School of Arts and Sciences, Lebanese American University
Beirut, Lebanon
{faisal.abukhzam}@lau.edu.lb

†School of Computer Science
University of Waterloo
Waterloo, Ontario, N2L 3G1, Canada
{aabdomou}@uwaterloo.ca

***Abstract*—Current generation supercomputers have over one million cores awaiting highly demanding computations and applications. An area that could largely benefit from such processing capabilities is naturally that of exact algorithms for $\mathcal{NP}$-hard problems. We propose a general implementation framework that targets highly scalable parallel exact algorithms for $\mathcal{NP}$-hard graph problems. We tackle the problems of efficiency and scalability by combining a fully decentralized dynamic load balancing strategy with special implementation techniques for exact graph algorithms. As a case-study, we use our framework to implement parallel algorithms for the VERTEX COVER and DOMINATING SET problems. We present experimental results that show notable improved running times on all types of input instances.**

Key words: Parallel Algorithms, Exact Algorithms, Vertex Cover, Dominating Set, Dynamic Load Balancing, Recursive Backtracking

I. INTRODUCTION

The constant increase in the number of processing elements per supercomputer motivates the development of parallel algorithms that can efficiently utilize such processing infrastructures. Efficiency in this context refers to both runtime speedups and scalability. Earlier clusters or computing platforms had a limited number of cores and did not impose high scalability issues. As this number increases, communication overheads can become the bottleneck of any parallel implementation that follows the classical master-slave approach.

On the other hand, the interest in exact algorithms has increased considerably during the last two decades. Thorough and deep studies have lead to constant development of exact algorithms with improved runtimes [8, 12, 11]. However, the practical aspect of proposed solutions and the possibility of exploiting available supercomputers using parallel implementations has been largely ignored thus far.

Most existing Algorithms for $\mathcal{NP}$-Complete graph problems follow the well-known branch-and-reduce paradigm. At the implementation level, this translates to search-tree based recursive backtracking algorithms. The search-tree size grows exponentially with either the size of the input instance n or some parameter k when the problem is fixed-parameter tractable ($\mathcal{FPT}$) [10]. Nevertheless, search-trees are good candidates for parallel decomposition. Given P cores, an embarrassingly parallel solution would divide a tree into P sub-trees and assign each to a separate core for sequential processing. This intuitive approach suffers from several drawbacks, including the obvious lack of load balancing.

We shall propose a general implementation framework that targets scalable parallel search-tree based exact algorithms for hard graph problems. As a case-study, we consider the optimization versions of two well-known problems: VERTEX COVER and DOMINATING SET.

II. BACKGROUND

The branch and reduce model is a popular algorithmic paradigm when dealing with $\mathcal{NP}$-hard problems. The approach often consists of applying a series of reduction rules followed by branching and pruning rules via a recursive search-tree based backtracking strategy. While reduction and pruning are rather deterministic procedures, branching usually results in two or more states that correspond to problem instances whose size is often smaller than the parent state.

When studying parallel implementations for search-tree based recursive backtracking algorithms, multiple aspects have to be considered in order to achieve desirable results. First, since we rely on the Message Passing Interface (MPI) [9], the communication overhead has to be minimized. Along with algorithmic techniques, a compact task representation scheme serves this purpose. Next, a dynamic load balancing strategy has to be selected. Several load balancing strategies have already been proposed in the literature [15]. Earlier work suggested static load balancing, which consists of dividing a search-tree into independent sub-trees and assigning each task to a core for sequential processing. It was quickly realized that, given the creation process of search-trees, certain tasks will be much "easier" than others. In the VERTEX COVER algorithm

for example, when branching on a vertex v, two instances of size $n-1$ and $n-|N(v)|$ respectively are produced. For large $|N(v)|$ values, the latter instance is likely to terminate earlier. This justifies the need for dynamic load balancing. Idle processing units (PUs) must dynamically be able to "help" other PUs under heavy load.

Known dynamic load balancing mechanisms range from receiver initiated (pull) [16] to sender initiated (push) or a hybrid of both. Most of these algorithms follow a master-worker architecture similar to the buffered work-pool approach presented in [5] where all of the communication burden is assigned to a single node (the master). In more recent work [1], high scalability was achieved using a load balancing strategy specifically designed for the VERTEX COVER problem. Using prior knowledge about generated instances, the authors in [1] proposed a dynamic master-slave approach where the node having the "hardest" instance is assigned as master.

As the number of cores available on supercomputers increases, the centralized approach is most likely to become the bottleneck of such algorithms. Thus, it is important to present a load balancing approach that is dynamic, scalable, and problem independent.

In an attempt to tackle all the difficulties discussed above, we present a decentralized load balancing strategy combined with a learning-based task management routine and appropriate implementation tricks that can be applied to most if not all search-tree based algorithms for graph problems. Several algorithms for VERTEX COVER and DOMINATING SET have been developed to compare and validate the efficiency of our methods.

A. Minimum Vertex Cover

Given a graph $G=(V,E)$, the optimization version of the VERTEX COVER problem searches for a set $C \subseteq V$ such that $|C|$ is minimized and the subgraph induced by $V \setminus C$ is edgeless. Even on relatively small instances, the VERTEX COVER problem, or VC for short, is sometimes very difficult to solve exactly using sequential algorithms.

The sequential algorithm for the parameterized version of VERTEX COVER having the fastest known worst-case behavior is due to [7] and runs in $\mathcal{O}(kn+1.2738^k)$ time. We convert the said algorithm to an optimization version by introducing simple modifications and excluding complex processing rules that require heavy maintenance operations. An alternative approach, would use the Maximum Independent Set algorithm which is the dual of the VC problem.

Some of the reduction/pruning rules in our algorithm require additional bookkeeping. In addition to the active solution size, we keep track of the best-so-far solution size in a global variable. By subtracting those two values, we compute k' which can be used similarly to the parameter k in the $\mathcal{FPT}$ version of the VC algorithm. During the search phase of the algorithm, a maximum-degree vertex v is selected and two branches (sub-problems) are generated; v is either placed in or excluded from C. In the latter case, $N(v)$ is forced to belong to any solution in order to cover all the edges incident on v. We refer to this branching strategy as the universal branching because it is common to a very large number of graph algorithms. Finally, two pruning rules are applied at every search-tree node. These rules are due to graph structural properties specific to the VERTEX COVER problem [6].

B. Minimum Dominating Set

Given an n-vertex graph $G=(V,E)$, the DOMINATING SET problem, hereafter DS, asks for a set $D \subset V$ such that $|D|$ is minimal and every vertex of G is either in D or adjacent to some vertex in D. The classical $\mathcal{NP}$-hard DS problem has obtained considerable interest in the area of graph optimization problems because of its tight relation to many other problems in different application domains. Until 2004, the best algorithm for DS was still the trivial $\mathcal{O}^*(2^n)$ enumeration.[1] In that same year, three algorithm were independently published breaking the $\mathcal{O}^*(2^n)$ barrier [12, 14, 17]. The best worst-case algorithm was presented by Grandoni with a running time in $\mathcal{O}^*(1.8019^n)$ [14]. Using measure-and-conquer, a bound of $\mathcal{O}^*(1.5137^n)$ was obtained on the running time of Grandoni's algorithm [11]. This was later improved to $\mathcal{O}^*(1.5063^n)$ in [18] and the current best worst-case algorithm can be found in [19] where a general algorithm for counting minimum dominating sets in $\mathcal{O}^*(1.5048^n)$ is also presented. For our experimental work, we implemented the algorithm of [11] where DS is solved by reduction to MINIMUM SET COVER (MSC). In the MSC problem, we are given a universe U of elements and a collection S of non-empty subsets of U. The goal is to find a subset $S' \subseteq S$ of minimum cardinality which covers U. DS is reduced to MSC by setting $U=V$ and $S=\{N[v] \mid v \in V\}$ where $N[v]$ denotes the closed neighborhood of v.

The DS algorithm also follows the universal branching rule. A set s of maximum cardinality is selected and search proceeds by creating two new sub-problems by either including s in the solution or deleting s. A small variation in this algorithm, when compared to VC, is that when all subsets of S have cardinality less than or equal to 2, branching is stopped and the problem is solved in polynomial time via a reduction to the Maximum Matching problem.

Both the VC and DS algorithms have several more preprocessing rules that help attain their fastest known worst-case behaviors [6, 8, 2, 11, 18, 19]. We neglect those rules for the same fundamental reason; Spending time on local search-tree node processing for avoiding worst-case behavior is not efficient in practical settings. In addition, parallel search-tree decomposition benefits more from fast search-tree node generation rather than local node inspections.

III. A DECENTRALIZED LOAD BALANCING STRATEGY

In the remainder of this paper, we present the main concepts, strategies, and implementation details of our parallel algorithms. First, we discuss the task representation scheme,

[1] Throughout this paper we use the modified big-Oh notation that suppresses all polynomially bounded factors. For functions f and g we say $f(n) \in \mathcal{O}^*(g(n))$ if $f(n) \in \mathcal{O}(g(n)poly(n))$, where $poly(n)$ is a polynomial.

task creation, and task management routines that enable fast task distribution in a decentralized model. Next, we introduce the notion of virtual network topology and conclude with experimental results illustrating the performance of the different versions of the implemented algorithms.

A. Task Representation

When considering a task representation scheme for message passing or task distribution, it is important to minimize the amount of data required to transfer. For the VC algorithm, we encode a task as follows:

- Integer value n for the number of active vertices.
- Integer value e for the number of active edges.
- Integer value $soltop$ for the number of vertices in the solution stack.
- Fixed size integer array for holding vertex degrees (i.e. the degree-vector).
- Variable size integer array for the solution stack.

We note that this representation can be reduced to only include the solution stack and stack size values. However, doing so would require extra pre-processing work before actual search can proceed (i.e: the degree-vector and some other values will have to be recomputed by going through the solution stack). Experimental results have showed that this reduction in task memory requirements is not beneficial unless some special attention is given to data structures used and the amount of operations executed prior to and during search.

The DS or MSC algorithm requires more bookkeeping thus a task is summarized by:

- Integer value s for the number of active sets.
- Fixed size integer array for the cardinality vector.
- Integer value e for the number of active elements.
- Fixed size integer array for the frequency vector.
- Integer value $soltop$ for the number of vertices in the solution stack.
- Variable size integer array for the solution stack.

B. Task Creation

Every PU maintains a task buffer of fixed size. After performing several experimental runs, setting the size of the task buffer to P, the number of PUs, outperformed all other trials. When the task buffer is too large, more time is spent creating tasks rather than solving them. In the opposite scenario, using a very small task buffer increases the communication overhead since the number of available tasks would not suffice to serve all requests. There are three questions to consider when developing task creation and management routines:

(1) When to create or circulate tasks?
(2) Where to create tasks?
(3) How to create tasks?

In our algorithms, the first item is controlled by user defined thresholds. A creation threshold, CT, associated to the task buffer, determines the level at which task creation should occur. In other terms, whenever the number of tasks in the buffer is less than or equal to CT, new tasks should be added. A starvation threshold (ST) and a distribution threshold (DT) control the task circulation flow. We adopt a receiver-initiated pull-based model. Distribution is only allowed when the number of tasks in the buffer is greater than or equal to DT and a task request has been received. Demand for tasks is initiated once the starvation level is reached.

Our decentralized load balancing strategy requires that tasks be created at each and every PU. A detailed description of this procedure is given in the next section. The most challenging aspect of a task creation module lies in item (3). The two desirable properties are: (i) task creation should occur at high levels in the search-tree and (ii) when hitting the creation threshold, new tasks should be produced as fast as possible. These two properties are contradictory and impose a tradeoff between task weight and creation speed. We define task weight as a function of the depth at which the task was generated in the search-tree and the number of remaining nodes in the graph (or instance size). Note that this definition is not problem dependent. The importance of these observations is two-fold; First, it is key to carefully select which branch to search and which to store as a task for future processing. As previously noted, the best-so-far solution size is used in both algorithms' pruning rules. Thus, the faster new and improved solutions can be found, the more efficient these rules become. Secondly, this definition of branch hardness allows for easier and effective task creation and management routines.

At early levels in the tree, tasks are heavy and require more processing time. Towards the bottom of the tree, tasks become light and are usually easy instances that can be solved quickly (delegating such tasks would introduce a communication overhead and should rather be solved in place). Careful tracing of recursive backtracking algorithms shows that a very high percentage of computational time is spent near the bottom of the search tree. To overcome this obstacle and promote heavy-weight task creation, it is possible to always add the left branch into the task buffer and proceed the search on the next branch. A justification of this choice is related to the depth-first search (DFS) nature of our algorithms. DFS always visits the left branch first. Therefore, creating a task as soon as the search starts and before we reach low levels in the tree helps sustaining tasks of heavier weight. However, light-weight task generation still occurs if no constraint is set on the minimum allowed task weight. We propose a dynamic learning mechanism that updates the minimum allowed task weight (MTW) as the search progresses.

C. Learning-based Task Management Routine

The initialization phase of our algorithms is carried out sequentially by a single PU. Reduction rules are applied until exhaustion, then P tasks are generated and broadcasted to all available PUs. In the search phase, each PU acts as a separate independent entity that is aware of its neighborhood. The communication infrastructure is a fully connected network. When a PU consumes all of its tasks, it can initiate a task request with any other PU selected pseudo-randomly from the pool of active workers. To avoid message repetition, a

starving PU does not request a task from the same PU twice when receiving a "NO TASK" response. In fact, randomness is restricted to PUs that have not been probed yet. When a task request has been sent to all PUs and no task was received, the requester can restart the process for a fixed number of times, say Q. When Q is exceeded, the requester enters the termination phase. To manage task requests and deliveries, every PU maintains a status vector about its neighborhood. PUs can be active, idle, or inactive. Any status change is broadcasted to all the participants. This approach might induce a large number of status notification messages but it eliminates the single-source bottleneck and scales easier as the number of PU increases. Alternatively, as we shall discuss later, the size of the neighborhood of each PU can be restricted by imposing a virtual topology.

As previously noted, generating heavy-weight tasks is a key feature for effective dynamic load balancing. We adopt a learning routine summarized as follows; Let $w(T)$ denote the weight of a given task T. For the VC algorithm, $w(T) = n'$ the number of remaining vertices in the graph. In the case of DS $w(T) = s'$ the number of unvisited sets in S. We denote the minimum task weight by $w_{min}(T) = 1$ and the maximum by $w_{max}(T)$ which is equal to the initial instance size minus 1. We divide $[w_{min}; w_{max}]$ into M weight range buckets $\{b_0, b_1, ..., b_M\}$ of equal length L. Each task is mapped into a certain bucket depending on its weight. If a task T_1 has weight $w_{max} - L < w(T_1) \leq w_{max}$, then it is mapped to the last bucket b_M (i.e $T_1 \in b_M$). Similarly, a task T_2 having $w_{max} - 2L < w(T_2) \leq w_{max} - L$ is mapped to bucket b_{M-1}. At early stages in the search, it is desirable and possible to spawn tasks belonging to bucket b_M. However, as the search progresses, such tasks are either depleted or hard to generate given that they are found at the top levels of the search-tree. We begin our search by only allowing task creation when $T \in b_M$. We use PU starvation as an indicator to update this constraint. When a PU has no more tasks and has requested one from every other PU without receiving any, then we set $T \in \{b_M \cup b_{M-1}\}$ as the new constraint and broadcast it. We say a complete pass has occurred. After each complete pass the process is repeated until the last allowed bucket is added. Our experiments have shown that tasks smaller then 10% of the initial problem instance can be considered as light-weight tasks. We set the values of M and L accordingly allowing only 10 buckets to be added in total (i.e. after 10 complete passes, the task creation constraint remains constant for all PUs). These variable are instance specific and may require slight adjustments on new input types. Heuristics may be used to automate the process of finding near optimal values.

Using the described algorithm, new solutions can be found by any PU and it is important that the new solution size be broadcasted to all participants. Doing so allows for more effective pruning and enables task cleaning. Given the updated solution size, each PU performs a local expired task cleaning operation before fulfilling new requests. Expired tasks are those whose current solution size is greater then or equal to the best-so-far value minus one.

IV. Virtual Network Topology

The underlying physical network topology of supercomputers can greatly affect the efficiency of parallel algorithms. Network design for supercomputers has been extensively studied in the literature [20, 21] and is outside the scope of this work. We note two key properties which are critical for successful designs:

(1) Diameter: defined as the maximum distance between two processors and can be used to estimate the lower bound for communication.
(2) Bisection width: useful for testing the ability of a network to deal with high traffic.

The decentralized load balancing method assumes a communication network consisting of a complete graph (fully connected network with diameter 1). However, most architectures do not satisfy this assumption and physical nodes could be separated by multiple hops and/or switches. Moreover, different supercomputers usually have completely different topologies and developing codes portable across all platforms would be a daunting task.

To reduce the number of message transmissions and still keep our codes portable, we introduced a variation to our model by enforcing a virtual network topology which is independent of the underlying hardware. A favorable topology consists of a regular graph having low diameter where each node is allowed to communicate with its neighbors only. One such graph is the well-known hypercube. In a hypercube of dimension d, each PU has d neighbors with a total of 2^d PUs. In addition to having small diameter, hypercubes allow for simple yet elegant bit operations for finding neighbors. We assign each PU an identifier of d bits. Flipping any of the d bits returns a neighbor of a given PU. The virtual topology slightly complicates termination detection. In particular, it is possible for idle PUs to terminate prematurely based on the information available from their direct neighborhood only. To address the issue we propose two different termination algorithms. The first, denoted by d-termination, works as follows: for a given idle node P, if all neighbors of P are idle, then P is $\frac{1}{d}$-terminated. If all neighbors of P are $\frac{1}{d}$-terminated and P is $\frac{1}{d}$-terminated, then P is $\frac{2}{d}$-terminated. In general, a given node is allowed to become inactive only when its closed neighborhood is $\frac{(d-1)}{d}$-terminated. Another approach is to force every $\frac{1}{d}$-terminated PU to ignore the virtual topology and switch back to a fully connected network before becoming inactive. The first termination algorithm does not induce a higher number of message transmissions but can cause idle times to increase proportionally to the amount of time it takes for a task to propagate through the network. The latter algorithm does not wait for task propagation but could require a higher number of task request message transmissions. We chose to implement the latter since it requires much less bookkeeping and is simpler to trace.

The impact of imposing such a virtual network in our algorithms varied with different instances. Runtimes were considerably better on some instances but remained almost

similar on others. However, when compared to the algorithms not using the topology, the number of message transmissions was consistently much lower for all test runs.

V. Experimental Results

To test our methods, we implemented three different versions of the Vertex Cover and Dominating Set algorithms:

(i) PVC1 and PDS1: The simple version of our parallel algorithms based on the decentralized communication model and the described task management procedure. The dynamic learning mechanism is excluded from this version and there is no explicit control over task creation.
(ii) PVC2 and PDS2: The complete version of our algorithm which includes the dynamic learning mechanism and considers each core as a separate processing unit.
(ii) PVC3 and PDS3: Same as the previous version but also includes the virtual network topology.

All codes were implemented in standard C using the MPI libraries and tested on a large supercomputer at Oak Ridge National Laboratory. Each reported execution time is the average of 4 independent runs on the same instance. To illustrate, we use a number of DIMACS graphs as input instances for the VC algorithms. For DS we generated several random graphs with varying densities.

TABLE I
PVC1, PVC2, and PVC3 average execution times.

Graph	P	PVC1	PVC2	PVC3
p_hat500_1.clq	2	48.3 min	44.85 min	51.75 min
p_hat500_1.clq	4	26.45 min	23 min	14.95 min
p_hat500_1.clq	8	19.2 min	9.2 min	6.9 min
p_hat500_1.clq	16	12.9 min	3.45 min	4.6 min
p_hat700_1.clq	2	$>$ 24 hrs	$>$ 24 hrs	$>$ 24 hrs
p_hat700_1.clq	4	991.3 min	871.7 min	600.3 min
p_hat700_1.clq	8	607.2 min	226.55 min	273.7 min
p_hat700_1.clq	16	342.7 min	151.8 min	133.4 min
p_hat700_1.clq	32	248.4 min	67.04 min	65.68 min
p_hat700_1.clq	64	370.3 min	31.41 min	32.76 min
p_hat700_1.clq	128		25.38 min	17.58 min
p_hat700_1.clq	256		12.78 min	9.48 min
p_hat700_1.clq	512		7.015 min	3.58 min
p_hat1000_2.clq	2	158.7 min	162.65 min	103.17 min
p_hat1000_2.clq	4	50.6 min	48.16 min	42.64 min
p_hat1000_2.clq	8	32.2 min	28.68 min	23.43 min
p_hat1000_2.clq	16	22.77 min	13.22 min	13.64 min
p_hat1000_2.clq	32	15.98 min	6.22 min	5.96 min
p_hat1000_2.clq	64	16.33 min	3.74 min	3.81 min

We start by comparing the three Vertex Cover algorithms on different DIMACS graphs by varying P the number of processing units. Results are shown in Table I. From Figure 1, we immediately notice that, when no constraints are set on minimum allowed task weights, scalability cannot be achieved. The PVC1 algorithm maintains acceptable speedups for up to 16 PUs only. On the other hand, linear (sometimes superlinear) speedups are possible when running the PVC2 pr PVC3 algorithms on the same instance. Similar behavior was spotted when running the different versions of the Dominating Set algorithm. Running times are shown in Table II and graphically in Figure 2.

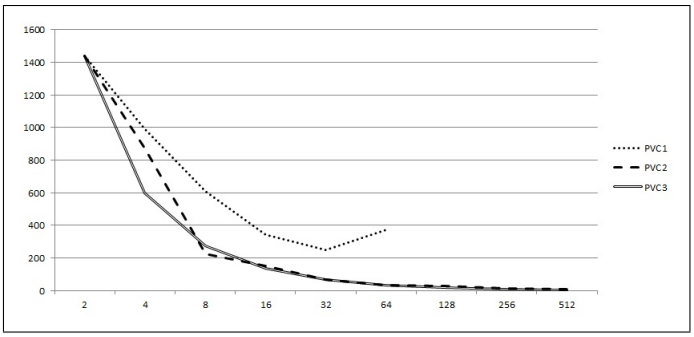

Fig. 1. PVC1 vs. PVC2 vs. PVC3 average execution times on p_hat700_1.clq.

TABLE II
PDS1, PDS2, and PDS3 average execution times.

Graph	P	PDS1	PDS2	PDS3
rgraph1	2	107.21 min	94.28 min	64.92 min
rgraph1	4	64.67 min	24.13 min	29.15 min
rgraph1	8	39.46 min	17.48 min	15.36 min
rgraph1	16	26.83 min	7.24 min	7.09 min
rgraph1	32	41.07 min	3.48 min	3.63 min
rgraph2	2	203.55 min	211.6 min	201.25 min
rgraph2	4	113.85 min	95.45 min	86.25 min
rgraph2	8	66.7 min	50.6 min	43.04 min
rgraph2	16	47.15 min	27.6 min	24.62 min
rgraph2	32	55.2 min	12.65 min	11.98 min
rgraph2	64		6.9 min	5.08 min
rgraph3	2	$>$ 24 hrs	$>$ 24 hrs	$>$ 24 hrs
rgraph3	4	$>$ 24 hrs	$>$ 24 hrs	$>$ 24 hrs
rgraph3	8	$>$ 24 hrs	$>$ 24 hrs	$>$ 24 hrs
rgraph3	16	1495 min	1577.8 min	1435.2 min
rgraph3	32	1144.25 min	803.85 min	761.3 min
rgraph3	64	1245.45 min	460 min	377.2 min
rgraph3	128		243.8 min	187.45 min
rgraph3	256		136.85 min	80.5 min
rgraph3	512		75.9 min	31.61 min
rgraph3	1024		38.3 min	19.78 min

VI. Conclusion

In this work, we presented a general decentralized load balancing strategy for parallel search-tree optimization combined with appropriate implementation techniques for improved efficiency. Preliminary experimental results have shown that both the Vertex Cover and Dominating Set algorithms achieve desirable speedups on all tested input instances. Throughout the testing phase of our algorithms we

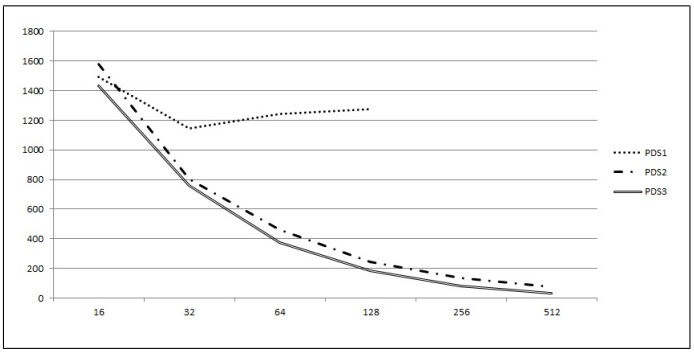

Fig. 2. PDS1 vs. PDS2 vs. PDS3 average execution times on rgraph3.

have spotted several areas for future improvements over the proposed approach. Our pseudo-random PU selection method for task requests can be replaced by a weight-based selection heuristic that forwards task requests to one of several PUs having a total task buffer weight greater then some predefined value. More importantly, given the crucial role of task weight in guaranteeing efficient load balancing, we shall investigate an index-based search-tree decomposition scheme that significantly reduces the cost of task buffer maintenance and task management overheads. Simply put, tasks would be created on-demand and at the highest unexplored branch in the search-tree.

REFERENCES

[1] D. Weerapurage, J. Eblen, G. Rogers, and M. Langston. (2011). "Parallel Vertex Cover: A case study in dynamic load balancing", in *Proceedings of the 9th Australasian Symposium on Parallel and Distributed Computing*.

[2] F. N. Abu-khzam, R. L. Collin, M. R. Fellows, M. A. Langston, W. H. Suters, and C. T. Symons. (2001). Kernelization algorithms for the vertex cover problem: Theory and experiments. [Online]. Available http://www.siam.org/meetings/alenex04/abstracts/F-Abu-Khzam.pdf

[3] F. N. Abu-Khzam, M. A. Langston, A. E. Mouawad and C. P. Nolan. A Hybrid Graph Representation for Recursive Backtracking Algorithms. Presented at Frontiers in Algorithmics, 2010, pp. 136–147. [Online]. Available http://www.springerlink.com/content/96u7250q37717834/

[4] F. N. Abu-Khzam, M. A. Langston, P. Shanbhag, and C. T. Symons, "Scalable parallel algorithms for fpt problems", *Algorithmica*, vol. 45, no. 3, pp. 269–284, 2006.

[5] F. N. Abu-Khzam, M. A. Rizk, D. A. Abdallah, and N. F. Samatova, "The buffered work-pool approach for search-tree based optimization algorithms", in *Proceedings of the 7th international conference on Parallel processing and applied mathematics*, 2008, pp. 170–179.

[6] J. Chen, I. A. Kanj, and W. Jia, "Vertex cover: Further observations and further Improvements", *Journal of Algorithms*, vol. 41, pp. 313–324, 2001.

[7] J. Chen, I. A. Kanj, and G. Xia, "Improved upper bounds for vertex cover", *Theoretical Computer Science*, vol. 411, pp. 3736–3756, 2010.

[8] J. Chen, L. Liu, and W. Jia, "Improvement on vertex cover for low-degree graphs", *Networks*, vol. 35, no. 4, pp. 253–259, 2000.

[9] J. J. Dongarra and D. W. Walker, "MPI: A message-passing interface standard", *International Journal of Supercomputing Applications*, 8(3/4), pp. 159–416. 3, 1994

[10] R. Downey and M. Fellows, *Parameterized Complexity*. Berlin: Springer, 1999.

[11] F. V. Fomin, F. Grandoni, and D. Kratsch, "Measure and conquer: Domination - a case study", in *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming*, London, UK, 2005, pp. 191–203.

[12] F. V. Fomin, D. Kratsch, J. Woeginger, and G. J. Woeginger, "Exact (exponential) algorithms for the dominating set problem", in *Proceedings of the 30th Workshop on Graph Theoretic Concepts in Computer Science*, 2004, pp. 245–256.

[13] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY: W. H. Freeman, 1979.

[14] F. Grandoni, "A note on the complexity of minimum dominating set", *Journal of Discrete Algorithms*, vol. 4, no. 2, pp. 209–214, 2006.

[15] V. Kumar, A. Y. Grama, and N. R. Vempaty, "Scalable load balancing techniques for parallel computers", *J. Parallel Distrib. Comput.*, vol. 22, no. 1, pp. 60–79, 1994.

[16] S. Patil and P. Banerjee, "A parallel branch and bound algorithm for test generation", in *Proceedings of the 26th ACM/IEEE Design Automation Conference*, New York, NY, 1989, pp. 339–343.

[17] B. Randerath and I. Schiermeyer, "Exact algorithms for minimum dominating Set", Zentrum für Angewandte Informatik Köln, Lehrstuhl Speckenmeyer, Tech. Rep., Apr. 2004.

[18] J. M. van Rooij and H. L. Bodlaender, "Design by measure and conquer, a faster exact algorithm for dominating set", In Symposium on Theoretical Aspects of Computer Science, Bordeaux, 2008, pp. 657–668.

[19] J. M. van Rooij, J. Nederlof, and T. C. van Dijk, "Inclusion/exclusion meets measure and conquer: Exact algorithms for counting dominating sets", Department of Information and Computing Sciences, Utrecht University, Tech. Rep. UU-CS-2008-043, 2008.

[20] S. Matic, "Emulation of Hypercube Architecture on Nearest-Neighbor Mesh-Connected Processing Elements", *IEEE Trans. Comput.*, vol. 39, no. 5, pp. 698–700, 1990.

[21] M. J. Rashti, J. Green, P. Balaji, A. Afsahi, and W. Gropp, "Multi-core and Network Aware MPI Topology Functions", *EuroMPI*, vol. 6960, pp. 50–60, 2011.