LEBANESE AMERICAN UNIVERSITY

A DECENTRALIZED LOAD BALANCING STRATEGY FOR PARALLEL SEARCH-TREE OPTIMIZATION

By

AMER E. MOUAWAD

A thesis

Submitted in partial fulfillment of the requirements for

the degree of Master of Science in Computer Science

School of Arts and Sciences

June 2010

LEBANESE AMERICAN UNIVERSITY

School of Arts and Sciences - Beirut Campus

Thesis Approval Form

Student Name: Amer Eid Abdo Mouawad I.D. #: 200400377

Thesis Title: A Decentralized Load Balancing Strategy for Parallel Search-Tree Optimization

Program: Master of Science in Computer Science

Department: Computer Science and Mathematics

School: School of Arts and Sciences - Beirut

Approved/Signed by:

Thesis Advisor: Dr. Faisal Abu-Khzam Signature: [redacted]

Committee Member: Dr. Samer Haber Signature: [redacted]

Committee Member: Dr. Ramzi Haraty Signature: [redacted]

Date: June 8, 2011

THESIS COPYRIGHT RELEASE FORM

LEBANESE AMERICAN UNIVERSITY

By signing and submitting this license, I (the author(s) or copyright owner) grant the Lebanese American University (LAU) the non-exclusive right to reproduce, translate (as defined below), and/or distribute my submission (including the abstract) worldwide in print and electronic format and in any medium, including but not limited to audio or video. I agree that LAU may, without changing the content, translate the submission to any medium or format for the purpose of preservation. I also agree that LAU may keep more than one copy of this submission for purposes of security, backup and preservation. I represent that the submission is my original work, and that I have the right to grant the rights contained in this license. I also represent that my submission does not, to the best of my knowledge, infringe upon anyone's copyright. If the submission contains material for which I do not hold copyright, I represent that I have obtained the unrestricted permission of the copyright owner to grant LAU the rights required by this license, and that such third-party owned material is clearly identified and acknowledged within the text or content of the submission. IF THE SUBMISSION IS BASED UPON WORK THAT HAS BEEN SPONSORED OR SUPPORTED BY AN AGENCY OR ORGANIZATION OTHER THAN LAU, I REPRESENT THAT I HAVE FULFILLED ANY RIGHT OF REVIEW OR OTHER OBLIGATIONS REQUIRED BY SUCH CONTRACT OR AGREEMENT. LAU will clearly identify my name(s) as the author(s) or owner(s) of the submission, and will not make any alteration, other than as allowed by this license, to my submission.

Name: Amer Eid Abdo Mouawad

Signature: [redacted] Date: June 7, 2011

PLAGIARISM POLICY COMPLIANCE STATEMENT

I certify that I have read and understood LAU's Plagiarism Policy. I understand that failure to comply with this Policy can lead to academic and disciplinary actions against me.

This work is substantially my own, and to the extent that any part of this work is not my own I have indicated that by acknowledging its sources.

Name: Amer Eid Abdo Mouawad

Signature:  Date: June 7, 2011

ACKNOWLEDGMENTS

I would like to thank all faculty members of the department of Computer Science and Mathematics at LAU for the continuous guidance and support. A special thanks to Dr. Faisal Abu-Khzam, my thesis advisor, for all the trust and motivation.

To my father...

A Decentralized Load Balancing Strategy For Parallel Search-Tree Optimization

Amer Eid Abdo Mouawad

Abstract

Current generation supercomputers have thousands of cores awaiting highly demanding computations and applications. One area that could largely benefit from such processing capabilities is clearly that of exact algorithms for $\mathcal{NP}$-hard problems. The interest in exact algorithms is more than fifty years old, but seems to have gained great momentum recently with the emergence of parameterized complexity and new exact algorithmic techniques. Moreover, given the proved limitation of polynomial-time approximation algorithms, many research fields witness greater need for accurate computations. Motivated by the above facts, we propose a general implementation framework that targets parallel exact algorithms for $\mathcal{NP}$-hard graph problems. In addition to efficiency, we tackle the problem of scalability by combining a decentralized dynamic load balancing strategy with new coding 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 and high scalability on all types of input instances.

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

Contents

List of Tables

List of Figures

Chapter One

1.1 Introduction

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. A notorious example is the partial model of the 120-cell graph which is a 4-regular graph (i.e. every vertex has degree 4) on 300 vertices and 600 edges [14]. On the other hand, current generation supercomputers have thousands of cores available for processing. This fact motivates the development of parallel algorithms that can efficiently utilize processing infrastructures provided by today's powerful supercomputers. 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 scalability issues. As the number of processing units increases, communication overheads can become the bottleneck of any parallel implementation that follows the classical master-slave approach.

The significance of exact algorithms and the need for accurate solutions has greatly increased in the last five decades. $\mathcal{NP}$-hard graph problems have probably been the main focus of the area of exact algorithms. Thorough studies have lead to the development of new algorithms with improved runtimes almost every year [2, 7, 4]. However, the practical aspect of proposed solutions and the possibility of exploiting available supercomputers using parallel implementations has received considerably less attention. Our interest in exact algorithms for hard graph problems

has been motivated by several factors such as:

- The increase in available processing power.
- The hardness of approximation of many $\mathcal{NP}$-Complete problems assuming the strongly believed hierarchical classification of computational problems [18].
- The need for accuracy in several application domains such as bioinformatics.
- The curse of double inaccuracies which arises when approximation algorithms are applied over simplified models of real-life problems (e.g. protein folding).

Most 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 (size of required solution) when the problem is fixed-parameter tractable (FPT) [20]. However, 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, mainly the lack of load balancing. As we shall also see, the efficiency of parallel implementations of such algorithms is affected by numerous variables.

In this work, we propose a general implementation framework that targets parallel exact algorithms for hard graph problems. As a case-study, we consider the optimization version of two well-known problems: Vertex Cover and Dominating Set. In addition to efficiency, we tackle the problem of scalability by combining a decentralized load balancing strategy with appropriate coding techniques. Our methods can easily be applied to other similar search-tree based problems.

1.2 Background

Most problems whose solution is a subset of the input can be solved by an adaptation of the branch-and-reduce model. Given any input instance, a series of reduction rules followed by branching rules are successively executed until a desired solution is found. Branching usually results in two or more states (problem instances) whose size is smaller than the parent state. Reduction rules are local modifications that reduce the size of a single search-state prior to the next branch. A third type of preprocessing rules that can have tremendous effects on runtimes are pruning rules. Pruning rules cut off certain branches in the search-tree and are desirable at early stages in the search. Pruning rules are usually ignored when analyzing algorithms' worst-case runtime since they cannot be guaranteed (otherwise would be suitable reduction rules). But, as our experimental results show, they perform very well in practice.

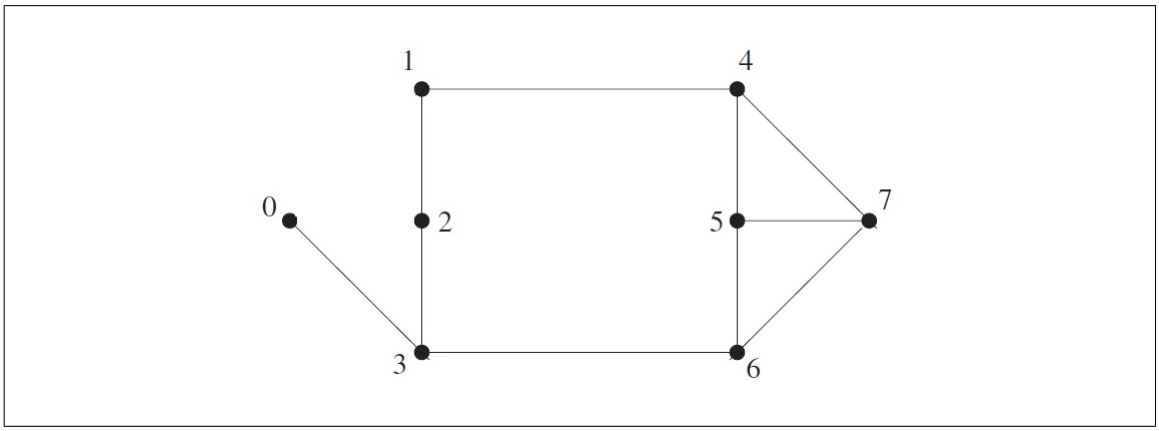

Figure 1: A sample graph, G

As an example, we consider the search-space generated when solving VC on graph G shown in Figure 1. For simplicity, we only adopt the universal high-degree branching rule which states that branching occurs on a vertex v where $|N(v)|$, the size of the open neighborhood of v, is maximal in the current version of G. The

first branch adds v to the solution C and deletes it from G along with its incident edges (thus producing a new version of G). The second branch adds $N(v)$ to the solution, since all edges have to be covered, and deletes the corresponding vertices and edges from G. In addition, we apply the degree-one reduction rule; For every vertex v having $|N(v)| = 1$, we add the neighbor of v into C. Figure 2 illustrates the generated search-tree. Reductions are represented by black arrows and branching scenarios (only one in this case) by solid lines. Clearly, $C = 3, 1, 7, 5$ is the desired solution. Since recursive backtracking algorithms follow a depth-first search strategy, the tree is visited from left to right. Knowing that a solution of size 4 has already been generated, the right sub-tree need not be explored because $|N(7)| + 2 = 5 \geq 4$. Programmatically, the previous observation can serve as a simple pruning rule. We shall present other applicable problem-specific pruning rules in later subsections.

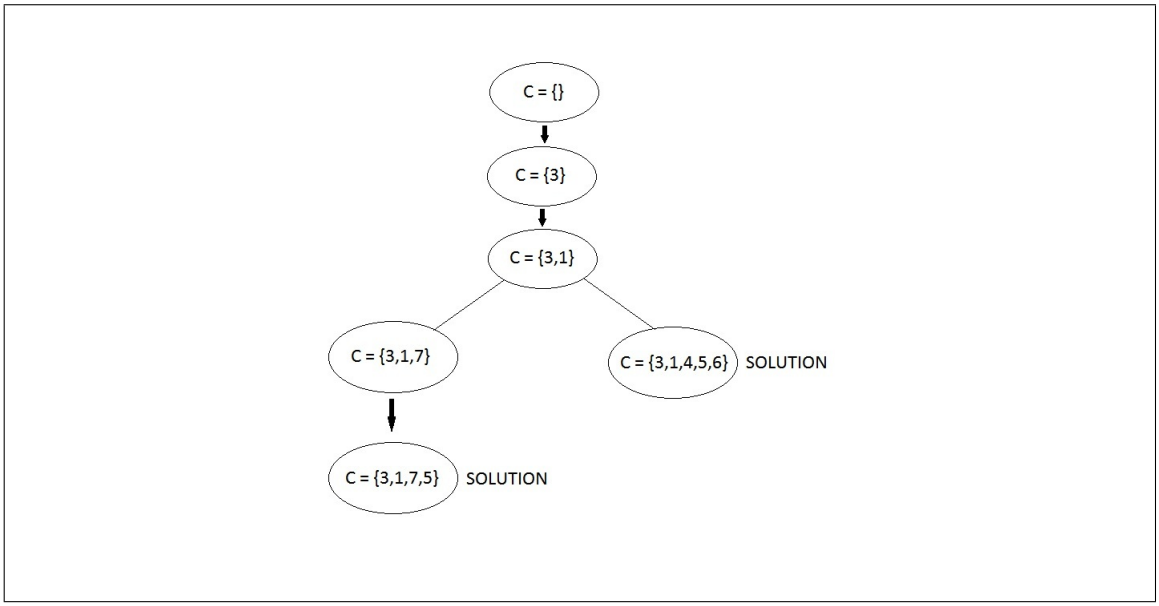

Figure 2: Search-tree example

Worst-case algorithms [1] are continually being revised and refined for better analytical runtimes. Improvements usually entail more sophisticated branching and/or

[1] We denote by "worst-case algorithm" the algorithm (for a given problem) with the best known asymptotic bound on its runtime.

reduction rules that require an increased number of polynomial-time "housekeeping" operations at every search-state. Due to the exponential number of search-tree nodes, these poly-time operations usually have a "butterfly effect" that has a negative impact on the efficiency of such algorithms. In fact, It was already shown that these worst-case algorithms are actually less efficient than simpler algorithms requiring less maintenance [16]. Appropriate data-structures that minimize the number of operations required during a search help reduce running times from days to hours.

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) [19], the communication overhead has to be minimized. 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 [23]. Earlier work suggested static load balancing which simply 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 from the previous section, the high-degree branch on vertex v produces two instances of size $n-1$ and $n-|N(v)|$ respectively. Clearly, for large $|N(v)|$ values, the latter is likely to terminate earlier. The high-degree rule is very common in exact algorithms and it 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) [24] 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 [27] where all of the communication burden is assigned to a single node (the master).

As the number of cores available on supercomputers increases, this centralized approach is certain to become the bottleneck of such algorithms. Thus, it is extremely important to present a load balancing approach that is both dynamic and scalable.

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.

Chapter Two

2.3 Vertex Cover Algorithm

The sequential algorithm for the parameterized version of Vertex Cover having the fastest known worst-case behavior is due to [28] 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 from [6] which is the dual of the VC problem.

2.3.1 Reduction Rules

The pendant-vertex rule. For every pendant, degree-one, vertex $v \in G$, we add its neighbor u to the cover since adding v cannot produce a smaller solution.

The high-degree rule. In the context of fixed parameter tractability, this rule states that any vertex v having $|N(v)| > k$ must be in the cover because otherwise all of its neighbors should be included and this violates the upper-bound k on the size of the cover. Applying this reduction rule in the optimization version of the algorithm requires some additional bookkeeping. We keep track of the current best known solution in $BESTSOL$ whose size is denoted by $BESTSOLTOP$ (size of the best solution found so far). During the search, SOL and $SOLTOP$ denote the current solution stack and its size respectively (SOL contains vertices added to the cover and does not necessarily represent a valid cover at every search-node). At any point in the search, we compute a value $k = BESTSOLTOP - SOLTOP - 1$ which can be used for processing the high-degree reduction rule. For any vertex v, if $SOLTOP + |N(v)| \geq BESTSOLTOP$ then v must be in the cover otherwise the

current best solution cannot be improved.

2.3.2 Branching Rules

In 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.

2.3.3 Pruning Rules

Two pruning rules are applied at every search-tree node. These rules are due to graph structural properties specific to the VERTEX COVER problem [1]. The first rule make sure the sum of degrees of the k vertices of highest degree is not less than the number of edges. The second rule counts the number of vertices having degree greater than zero and validates that this number is greater than $k*(1+(0.5*maxd))$. Here, k is equal to $BESTSOLTOP - SOLTOP - 1$ and $maxd$ is the maximum degree found in the graph. Whenever one of these two conditions is false, the corresponding sub-tree is pruned and the current best solution size is returned.

2.4 Dominating Set Algorithm

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.[2] In that same year, three algorithm were independently published breaking the $\mathcal{O}^*(2^n)$ barrier [7, 8, 10]. The best worst-case algorithm was presented by Grandoni with a running time in $\mathcal{O}^*(1.8019^n)$ [8]. Using measure-and-conquer, a bound of $\mathcal{O}^*(1.5137^n)$ was obtained on the running time of Grandoni's algorithm [4]. This was later improved to $\mathcal{O}^*(1.5063^n)$ in [13] and the current best worst-case algorithm can be found in [12] 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 [4] 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.

2.4.1 Reduction Rules

The dominated-set rule. If there are two distinct sets P and Q in S and $P \subseteq Q$, then there is a minimum set cover which does not contain P.

The frequency-one rule. If there is an element $u \in U$ which belongs to a unique set $P \in S$, then P belongs to every set cover.

2.4.2 Branching Rules

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

[2]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.

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.

2.4.3 Pruning Rules

The only pruning rule applied to the DS algorithm can be stated as follows: If the sum of cardinalities of the k sets of greatest cardinality is greater than the number of elements, the current best solution is returned. The value of k is defined similarly to that of the VC algorithm.

2.5 Excluded Reduction Rules

Both the Vertex Cover and Dominating Set algorithms have several more preproccessing rules that help attain their fastest known worst-case behaviors. However, the cost of these rules varies from linear to polynomial time and they only apply to special instances that are very rarely encountered in real data. Excluding such expensive rules greatly reduces preprocessing time and renders simple algorithms more efficient since the exponential "butterfly effect" is avoided.

In FPT terminology, reduction rules are also referred to as kernelization procedures that can guarantee a problem core (kernel) whose size depends on k instead of n. For the Vertex Cover problem, kernelization methods include degree-two vertex folding described in [1, 2], crown-decomposition [15], and other linear programming techniques [17]. All of these reduction rules are excluded from the optimization version of our VC algorithm (even though they can be applied) based on results from [16] and knowing that parallel search-tree decomposition benefits more

from fast search-tree node generation rather than local node inspections. DOMINATING SET does not belong to the class of FPT problems but has nonetheless numerous complex reduction rules [4, 13, 12] that we neglect for the same fundamental reason; Spending time on local search-tree node processing for avoiding worst-case behavior is not efficient in practical settings.

Chapter Three

3.6 Parallel Search-Tree Decomposition

In this section, we present the main concepts, strategies, and implementation details of our parallel algorithms. First we discuss the task representation scheme and task management routine that enable fast task distribution in parallel search-tree decomposition. Then, a new decentralized load balancing strategy is described. The purpose of this decentralized model is to overcome the bottleneck of master-slave architectures that cannot achieve scalability as the number of cores considerably increases. Finally, we explain the termination detection module and conclude by providing some remarks on the use of multi-threading combined with message passing as well as replacing classical data structures with the hybrid data structure for graph representation from [16].

3.6.1 Task Management

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 remaining vertices.
- Integer value e for the number of remaining 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. We address these implementation details in later sub-sections. The DS or MSC algorithm requires more bookkeeping thus a task is summarized by:

- Integer value s for the number of remaining sets.
- Fixed size integer array for the cardinality vector.
- Integer value e for the number of remaining 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.

Both algorithms are recursive backtracking algorithms and follow a depth-first search strategy. Both generated search-trees are binary trees visited from left to right. In the VC case, the left branch places the selected vertex in the solution and generates harder tasks since the right branch usually produces smaller instances. Thus, we denote this left branch by the "hard branch". To keep the same topology in the DS algorithm, we let the left branch be the case where a set is deleted (excluded from the solution). So now we refer to the left branch in both algorithms as the hard branch. Hardness, in this context, refers to how deep in tree the search has

to proceed before backtracking occurs or a solution is found. The importance of these observations is two-fold; First, it is very important to carefully select which branch to search and which to store as a task for future processing. As previously noted, the $BESTSOLTOP$ value 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.

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 requesting PUs.

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). At early levels in the tree, tasks are heavy and require more processing time.

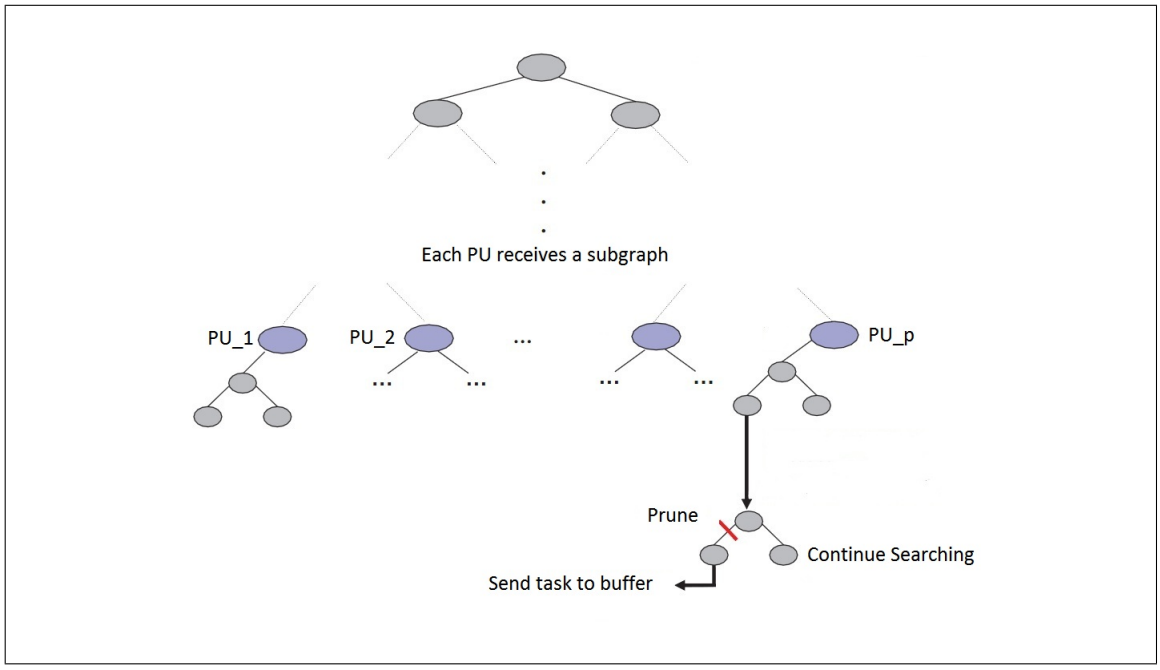

Figure 3: Task creation

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 guarantee heavy-weight task creation, we designate the hard branch (left branch) as task candidate. So, whenever task generation is required, the left branch is placed in the buffer and search proceeds on the right branch. Another 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. Even with all the described techniques, light-weight task creation 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 (more details later). The tradeoff between task weight and creation speed is due to the fact that task creation is restricted to higher levels in the search-tree while most of the processing time is spent in the lower levels. Thus when tasks are needed, it might take a while before an acceptable creation spot is reached.

3.6.2 Decentralized Load Balancing

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 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 a limited number of times. This limitation is related to termination detection and is discussed in the next section. 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. Even though this decentralized approach might induce a larger number of message transmissions when compared to a master-slave approach, it eliminates the single-source bottleneck and scales easier as the number of PU increases.

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 has weight $w_{max} - L < w(T) \leq w_{max}$, then it is mapped to the last bucket b_M. 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. An automated process for finding near optimal values can be developed but has been left for future work due to time restrictions. A hypothetical search-tree division into buckets is shown in Figure 4.

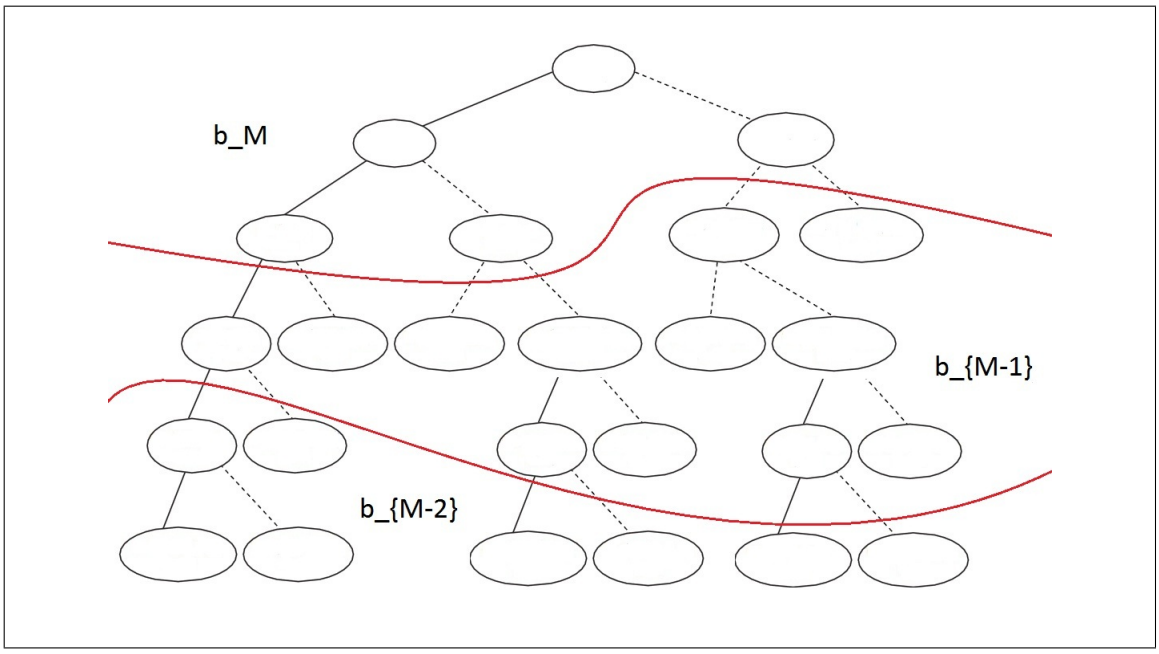

Figure 4: Hypothetical bucket division

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.

3.6.3 Termination Detection

Termination detection in a decentralized model is not as straightforward as in the master/worker case. In addition to the problem of messages in transit, no termination signal can be issued to all PUs simultaneously. The fact that we are considering the optimization versions of the VC and DS problems combined with the use of non-blocking send/receive messages also hardens the challenge. Additional book keeping and message transmissions are required to address all aspects of the problem. Each PU holds a status vector that saves the current status of all other workers in the pool; -1 for dead, 0 for idle, and 1 for active PUs. Once 10 complete passes have occurred, all PUs reset their counters and every pass becomes a termination pass. After two termination passes, a PU switches to the idle state and broadcasts a notification. The following rules apply given one or more PUs are idle:

- Idle PU keeps listening for incoming messages.
- No task requests are sent to or from idle PUs.
- A "NO TASK" reply is sent from idle PUs when receiving a task request that was still in transit before the notification reached all recipients.

When all PUs reach the idle state, a one minute waiting time is forced to validate that there are no messages in transit. Finally, termination can safely proceed.

3.7 Implementation

3.7.1 Message Passing vs. Multi-Threading

Most of today's supercomputers consist of a network of inter-connected nodes with one or more CPUs and multiple cores. In MPI terminology, a processing unit refers

to a single core. When designing parallel algorithms to run on such infrastructures two approaches are possible:

(1) Message passing combined with multi-threading where each node represents a single processing units and cores are utilized by several working threads.

(2) Message passing alone which emphasizes each core as a separate processing unit able of communicating with cores on other nodes or on the same node.

The use of multi-threading reduces the memory consumption of each node since only one copy of the graph is needed (per node). However, locking mechanisms are required to protect mutual access to critical sections of the code. To evaluate the effectiveness of both methods, we implemented two versions of our algorithms. Experimental results, provided in the last section, seem to favor the single-thread approach. A reasonable conclusion given the extra computational cost introduced by locking as well as the lack of programmer control over thread-to-core allocation.

3.7.2 Hybrid vs. Classical Data Structures

It is now a well-known fact that search-tree based recursive backtracking algorithms, such as those studied in this work, become impractical as the number of actions (i.e. operations) required for branching decisions increases. When backtracking, every executed action or modification has to be undone which almost doubles the cost per operation. From a coding standpoint, this means that avoiding a single loop statement (or reducing its complexity) can have big effects on runtimes. In fact, these propositions have been verified in [16]. The authors presented a new hybrid data structure for storing graphs in memory. The main advantage of the proposed representation compared to classical representations is that it combines the advantage of $\mathcal{O}(1)$ adjacency-queries in adjacency-matrices with the advantage of efficient

neighborhood traversal in adjacency-lists. The cost of undo operations is also minimized by efficiently trading space for time and enabling implicit backtracking. Using this data structure, the authors in [16] have shown that running times of the same algorithm can be consistently reduced, sometimes from days to hours.

The hybrid graph representation was designed with sequential algorithms in mind and is dynamically modified during search. Migrating this technique for parallel execution requires a reset operation before any task can be carried on. A copy of the original degree vector has to be stored on every PU and the current graph layout is reconstructed by a single pass through the solution vector. This seemingly simple operation runs in $\mathcal{O}(n)$ time and can render the parallel use of the hybrid data structure less practical. We implemented two additional different versions of the parallel VC and DS algorithms using the hybrid representation and ran several test simulations. Experimental results confirm our assumptions. When task creation is restricted to heavy tasks only runtimes are improved. If medium or light weight task creation is allowed, the $\mathcal{O}(n)$ overhead nullifies the advantages of hybrid graph representation. Results and variables used in our experiments are provided in the next section.

Chapter Four

4.8 Experimental Setup

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

(i) MPI_VC_SIMPLE and MPI_DS_SIMPLE: The simple version of our parallel algorithm 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) MPI_VC_LEARNING and MPI_DS_LEARNING: The complete version of our algorithm which includes the dynamic learning mechanism and considers each core as a separate processing unit.

(iii) MPIPT_VC_LEARNING and MPIPT_DS_LEARNING: Similar to (ii) but realized using both message passing and multi-threading.

All codes were implemented in standard C using the MPI and PTHREADS libraries. Experiments were run on a 64 nodes cluster with 2 cores per node (i.e. a total of 128 processing units). 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. In addition, real DS instances for biological problems were obtained from the Gene Expression Omnibus (GEO) data-sets available at http://www.ncbi.nlm.nih.gov/. The raw data (SOFT) files were transformed into simple unweighted graphs using Pearsons coefficients and appropriate thresholding. The threshold value used for each graph appears in the file extension.

4.9 Preliminary Results

We start by comparing the three VERTEX COVER algorithms on a single graph p_hat500_3.clq with 500 vertices and 30950 edges by varying P the number processing units. Results are shown in Table 1.

Table 1: Comparison of different VC algorithms on p_hat500_3.clq.

P	MPI_VC_SIMPLE	MPI_VC_LEARNING	MPIPT_VC_LEARNING
2	751 min	769 min	892 min
4	457 min	427 min	701 min
8	322 min	238 min	613 min
16	270 min	169 min	559 min
32	271 min	108 min	506 min
64	295 min	83 min	427 min
128	312 min	62 min	294 min

From Figure 5, we immediately notice that, when no constraints are set on minimum allowed task weights, scalability cannot be achieved. The MPI_VC_SIMPLE algorithm maintains acceptable speedups for up to 16 PUs only. On the other hand, almost linear speeupds are possible when running the MPI_VC_LEARNING algorithm on the same instance. As for the MPIPT_VC_LEARNING algorithm, no conclusive results are obtained but it clearly does not outperform the single-threaded version in terms of execution speed. To validate these results, we run the MPI_VC_LEARNING algorithm on two additional instances. Execution times are given in Tables 2 and 3. Graphical illustrations are shown in Figure 6. Similar behavior was spotted when running the different versions of the DOMINATING SET algorithm. Running times on random graphs and real data-sets are shown in Tables 4, 5 and 6, 7 respectively.

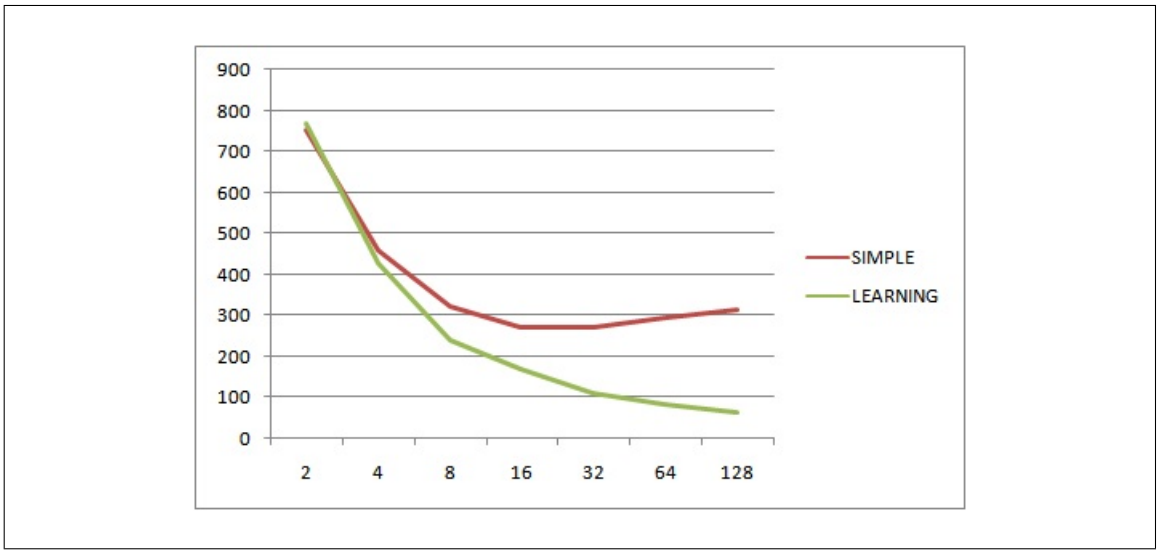

Figure 5: Average execution times on p_hat500_3.clq.

Table 2: MPI_VC_LEARNING average execution times on p_hat500_1.clq.

Graph	$\|V\|$	$\|E\|$	$\|C\|$	P	LEARNING
p_hat500_1.clq	500	31569	450	2	522 min
p_hat500_1.clq	500	31569	450	4	417 min
p_hat500_1.clq	500	31569	450	8	300 min
p_hat500_1.clq	500	31569	450	16	208 min
p_hat500_1.clq	500	31569	450	32	112 min
p_hat500_1.clq	500	31569	450	64	79 min
p_hat500_1.clq	500	31569	450	128	54 min

Table 3: MPI_VC_LEARNING average execution times on p_hat1000_2.clq.

Graph	$\|V\|$	$\|E\|$	$\|C\|$	P	LEARNING
p_hat1000_2.clq	1000	244799	946	2	$>$ 2400 min
p_hat1000_2.clq	1000	244799	946	4	2334 min
p_hat1000_2.clq	1000	244799	946	8	1459 min
p_hat1000_2.clq	1000	244799	946	16	858 min
p_hat1000_2.clq	1000	244799	946	32	613 min
p_hat1000_2.clq	1000	244799	946	64	474 min
p_hat1000_2.clq	1000	244799	946	128	401 min

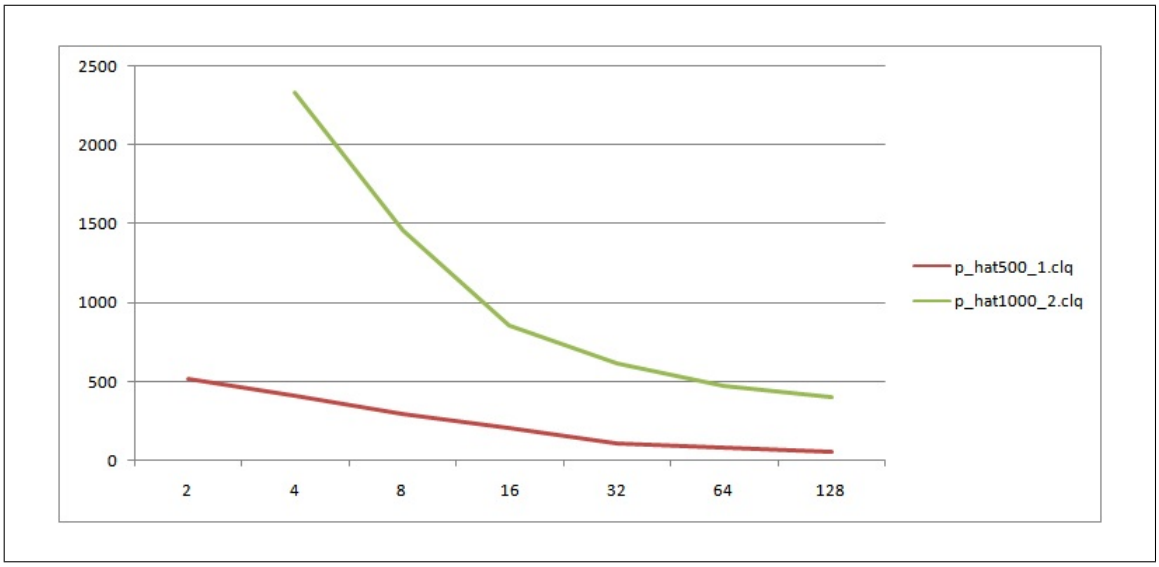

Figure 6: MPI_VC_LEARNING average execution times.

Table 4: MPI_DS_LEARNING average execution times on random graph 1.

Graph	$\|V\|$	$\|E\|$	$\|D\|$	P	LEARNING
rgraph5	150	1500	11	2	272 min
rgraph5	150	1500	11	4	207 min
rgraph5	150	1500	11	8	143 min
rgraph5	150	1500	11	16	99 min
rgraph5	150	1500	11	32	87 min
rgraph5	150	1500	11	64	64 min
rgraph5	150	1500	11	128	54 min

Table 5: MPI_DS_LEARNING average execution times on random graph 2.

Graph	$\|V\|$	$\|E\|$	$\|D\|$	P	LEARNING
rgraph14	250	12000	5	2	316 min
rgraph14	250	12000	5	4	280 min
rgraph14	250	12000	5	8	263 min
rgraph14	250	12000	5	16	204 min
rgraph14	250	12000	5	32	181 min
rgraph14	250	12000	5	64	114 min
rgraph14	250	12000	5	128	84 min

Table 6: MPI_DS_LEARNING average execution times on GDS3221.94.

Graph	$\|V\|$	$\|E\|$	$\|D\|$	P	LEARNING
GDS3221.94	8517	131498	2042	2	99 min
GDS3221.94	8517	131498	2042	4	80 min
GDS3221.94	8517	131498	2042	8	56 min
GDS3221.94	8517	131498	2042	16	35 min
GDS3221.94	8517	131498	2042	32	24 min
GDS3221.94	8517	131498	2042	64	26 min
GDS3221.94	8517	131498	2042	128	34 min

Table 7: MPI_DS_LEARNING average execution times on GDS3221.93.

Graph	$\|V\|$	$\|E\|$	$\|D\|$	P	LEARNING
GDS3221.93	11065	315488	2427	2	285 min
GDS3221.93	11065	315488	2427	4	211 min
GDS3221.93	11065	315488	2427	8	175 min
GDS3221.93	11065	315488	2427	16	170 min
GDS3221.93	11065	315488	2427	32	123 min
GDS3221.93	11065	315488	2427	64	107 min
GDS3221.93	11065	315488	2427	128	77 min

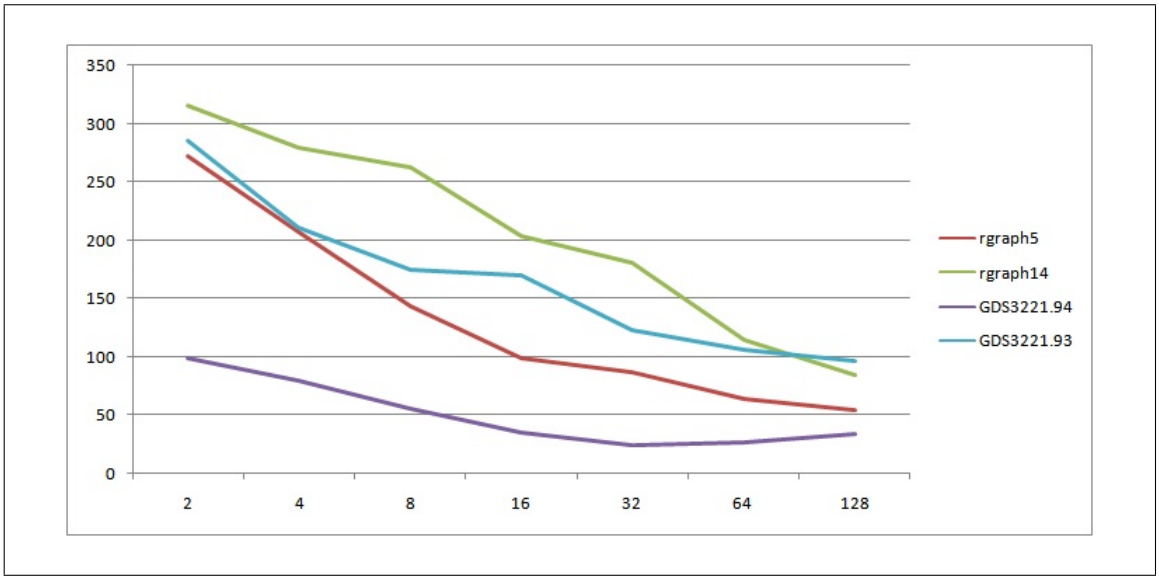

Figure 7: MPI_DS_LEARNING average execution times.

Chapter Five

5.10 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 instance inputs. Scalability was attained as the number of involved processing units increased but testing on larger supercomputers is still required for confirming scalability over thousands of cores. Throughout the testing phase of our algorithms we 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] J. Chen, I. A. Kanj, and W. Jia, "Vertex cover: Further observations and further Improvements", *Journal of Algorithms*, vol. 41, pp. 313–324, 2001.

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

[3] L. Engebretsen and J. Holmerin, "Clique is hard to approximate within $n^{1-o(1)}$", in *Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, London, UK, 2000, pp. 2–12.

[4] 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.

[5] F. V. Fomin, F. Grandoni, and D. Kratsch, "Solving connected dominating set faster than 2^n", *Algorithmica*, vol. 52, no. 2, pp. 153–166, 2008.

[6] F. V. Fomin, F. Grandoni, and D. Kratsch, "Measure and conquer: A simple $O(2^{0.288n})$ independent set algorithm", in *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, New York, NY, USA, 2006, pp. 18–25.

[7] 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.

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

[9] J. Hastad, "Clique is hard to approximate within $n^{(1-\epsilon)}$", *Acta Mathematica*, pp. 627–636, 1996.

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

[11] J. M. van Rooij and H. L. Bodlaender, "Exact algorithms for edge domination", Department of Information and Computing Sciences, Utrecht University, Tech. Rep. UU-CS-2007-051, 2007.

[12] 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.

[13] 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.

[14] S. Debroni, E. Delisle, W. Myrvold, A. Sethi, J. Whitney, J. Woodcock, P. W. Fowler, B. De La Vaissire, and M. Deza. Maximum independent sets of the 120-cell and other regular polyhedral. ARS MATHEMATICA CONTEMPORANE [Online]. Available http://www.liga.ens.fr/vdeza/withfowler/120-cell_2010.pdf

[15] 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

[16] Faisal N. Abu-Khzam, Michael A. Langston, Amer E. Mouawad and Clinton 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/

[17] 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.

[18] I. M. Bomze, M. Budinich, P. M. Pardalos, and M. Pelillo, "The maximum clique problem", *Handbook of Combinatorial Optimization*, pp. 1–74, 1999.

[19] 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

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

[21] 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.

[22] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI (2nd ed.): Portable parallel programming with the message-passing interface*. Cambridge, MA: MIT Press, 1999

[23] 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.

[24] 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.

[25] A. Reinefeld, "Scalability of massively parallel depth-first search", in *DIMACS Workshop*, 1994, pp. 305–322.

[26] A. Reinefeld and V. Schnecke, "Work load balancing in highly parallel depth first search", in *Scalable High Performance Computing Conference*, 1994, pp. 773–780.

[27] 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.

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