On Scalable Parallel Recursive Backtracking

Faisal N. Abu-Khzam[a,*], Khuzaima Daudjee[b,1], Amer E. Mouawad[b,1], Naomi Nishimura[b,1]

[a] *Department of Computer Science and Mathematics*
Lebanese American University
Beirut, Lebanon
[b] *David R. Cheriton School of Computer Science*
University of Waterloo
Waterloo, Ontario, N2L 3G1, Canada

Abstract

Supercomputers are equipped with an increasingly large number of cores to use computational power as a way of solving problems that are otherwise intractable. Unfortunately, getting serial algorithms to run in parallel to take advantage of these computational resources remains a challenge for several application domains. Many parallel algorithms can scale to only hundreds of cores. The limiting factors of such algorithms are usually communication overhead and poor load balancing. Solving NP-hard graph problems to optimality using exact algorithms is an example of an area in which there has so far been limited success in obtaining large scale parallelism. Many of these algorithms use recursive backtracking as their core solution paradigm. In this paper, we propose a lightweight, easy-to-use, scalable approach for transforming almost any recursive backtracking algorithm into a parallel one. Our approach incurs minimal communication overhead and guarantees a load-balancing strategy that is implicit, i.e., does not require any problem-specific knowledge. The key idea behind our approach is the use of efficient traversal operations on an indexed search tree that is oblivious to the problem being solved. We test our approach with parallel implementations of algorithms for the well-known Vertex Cover and Dominating Set problems. On sufficiently hard instances, experimental results show nearly linear speedups for thousands of cores, reducing running times from days to just a few minutes.

Keywords: parallel algorithms, recursive backtracking, load balancing, vertex cover, dominating set

*Corresponding author.
Email addresses: faisal.abukhzam@lau.edu.lb (Faisal N. Abu-Khzam), kdaudjee@uwaterloo.ca (Khuzaima Daudjee), aabdomou@uwaterloo.ca (Amer E. Mouawad), nishi@uwaterloo.ca (Naomi Nishimura)

[1]Research supported by the Natural Science and Engineering Research Council of Canada.

1. Introduction

Parallel computation is becoming increasingly important as performance levels out in terms of delivering parallelism within a single processor due to Moore's law. This paradigm shift means that to attain speedup, software that implements algorithms that can run in parallel on multiple processors/cores is required. Today we have a growing list of supercomputers with tremendous processing power. Some of these systems include more than a million computing cores and can achieve up to 30 Petaflop/s. The constant increase in the number of processors/cores per supercomputer motivates the development of parallel algorithms that can efficiently utilize such processing infrastructures. Unfortunately, migrating known serial algorithms to exploit parallelism while maintaining scalability is not straightforward. The overheads introduced by parallelism are very often hard to evaluate, and fair load balancing is possible only when accurate estimates of task "hardness" or "weight" can be calculated on-the-fly. Providing such estimates usually requires problem-specific knowledge, rendering the techniques developed for a certain problem useless when trying to parallelize an algorithm for another.

As it is not likely that polynomial-time algorithms can be found for NP-hard problems, the search for fast deterministic algorithms could benefit greatly from the processing capabilities of supercomputers. Researchers working in the area of exact algorithms have developed algorithms yielding lower and lower running times [1, 2, 3, 4, 5]. However the major focus has been on improving the asymptotic worst-case behavior of algorithms. The practical aspects of the possibility of exploiting parallel infrastructures has received much less attention.

Most existing exact algorithms for NP-hard graph problems follow the well-known branch-and-reduce paradigm. A branch-and-reduce algorithm searches the complete solution space of a given problem for an optimal solution. Simple enumeration is usually prohibitively expensive due to the exponentially increasing number of potential solutions. To prune parts of the solution space, an algorithm uses reduction rules derived from bounds on the function to be optimized and the value of the current best solution. The reader is referred to Woeginger's excellent survey paper on exact algorithms for further details [6]. At the implementation level, branch-and-reduce algorithms translate to search-tree-based recursive backtracking algorithms. The search tree size usually grows exponentially with either the size of the input instance n or some integer parameter k when the problem is fixed-parameter tractable [7].

Nevertheless, search trees are good candidates for parallel decomposition. While most divide-and-conquer methods for parallel algorithms aim at partitioning a problem instance among the cores, we partition the search space of the problem instead. Given c cores or processing elements, a brute-force parallel solution would divide a search tree into c subtrees and assign each subtree to a separate core for sequential processing. One might hope to thus reduce the overall running time by a factor of c. However, this intuitive approach suffers from several drawbacks, including the obvious lack of load balancing.

Even though our focus is on NP-hard graph problems, we note that recursive

backtracking is a widely-used technique for solving a very long list of practical problems. This justifies the need for a general strategy to simplify the migration from serial to parallel algorithms. One example of a successful parallel framework for solving different types of problems is MapReduce [8]. The success of the MapReduce model can be attributed to its simplicity, transparency, and scalability, all of which are properties essential for any efficient parallel algorithm. In this paper, we propose a simple, lightweight, scalable approach for transforming almost any recursive backtracking algorithm into a parallel one with minimal communication overhead and a load balancing strategy that is implicit, i.e., does not require any problem-specific knowledge. The key idea behind our approach is the use of efficient traversal operations on an indexed search tree that is oblivious to the problem being solved. To test our approach, we implement parallel exact algorithms for the well-known Vertex Cover and Dominating Set problems. Experimental results show that for sufficiently hard instances, we obtain nearly linear speedups on at least 32,768 cores.

2. Preliminaries

Typically, a recursive backtracking algorithm exhaustively explores a search tree T using depth-first search traversal. Each node of T (a *search node*) maintains some data structures required for completing the search. We denote a search node by $N_{d,p}$, where d is the depth of $N_{d,p}$ in T and p is the position of $N_{d,p}$ in the left-to-right ordering of all search nodes at depth d. The root of T is thus $N_{0,0}$. We use $T(N_{d,p})$ to denote the subtree rooted at node $N_{d,p}$. We say T has *branching factor* b if every search node has at most b children. A generic serial recursive backtracking algorithm, Serial-RB, is given in Figure 1.

```
1:  procedure SERIAL-RB(N_{d,p})
2:      if (ISSOLUTION(N_{d,p})) then
3:          best_so_far ← N_{d,p};
4:      if (ISLEAF(N_{d,p})) then
5:          Backtrack;                                  ▷ undo operations
6:      p' ← 0;
7:      while HASNEXTCHILD(N_{d,p}) do
8:          N_{d+1,p'} ← GETNEXTCHILD(N_{d,p});
9:          SERIAL-RB(N_{d+1,p'});
10:         p' ← p' + 1;
```

Figure 1: The Serial-RB algorithm (here p' denotes the position of a search node in the left-to-right ordering of the node and its siblings)

As an example, consider the problem of finding a minimum set of vertices $S \subset V$ of a graph $G = (V, E)$ such that the graph induced by $V \setminus S$ is a forest, i.e. a graph with no cycles. A possible implementation of Serial-RB

which solves this problem, also known as the MINIMUM FEEDBACK VERTEX SET problem, is as follows. Every search node maintains a graph $G' = (V', E')$ and a solution set S'. We use $N_{d,p}(G')$ and $N_{d,p}(S')$ to denote the graph and the solution set at node $N_{d,p}$, respectively. At $N_{0,0}$, we have $N_{0,0}(G') = G$ and $N_{0,0}(S') = \emptyset$. The ISSOLUTION($N_{d,p}$) function returns true whenever the graph induced by $N_{d,p}(V') \setminus N_{d,p}(S')$ is a forest and $|N_{d,p}(S')| < |best\_so\_far(S')|$, i.e the size of the smallest solution found so far. The ISLEAF($N_{d,p}$) function returns true when the current branch cannot lead to any better solutions (e.g., whenever $|N_{d,p}(S')| \geq |best\_so\_far(S')|$). Finally, to generate the children of a search node, we simply find a cycle in $N_{d,p}(G')$ and for each vertex v in that cycle we get a new search node $N_{d+1,p'}$, where $N_{d+1,p'}(S') = N_{d,p}(S') \cup \{v\}$ and $N_{d+1,p'}(G')$ is obtained by deleting v and all the edges incident on v from $N_{d,p}(G')$. In terms of exact algorithms [6], GETNEXTCHILD corresponds to the implementation of *branching rules* and ISLEAF implements *pruning rules*. If we let G be a graph consisting of two triangles sharing an edge, then Figure 2 shows one possible search tree generated by the described algorithm. Even though $N_{1,1}(G')$ is not acyclic, the children of $N_{1,1}$ will be pruned. This follows from the fact that, in a serial execution, $N_{1,0}(S')$ is a solution of size one and hence ISLEAF($N_{1,1}$) would return true.

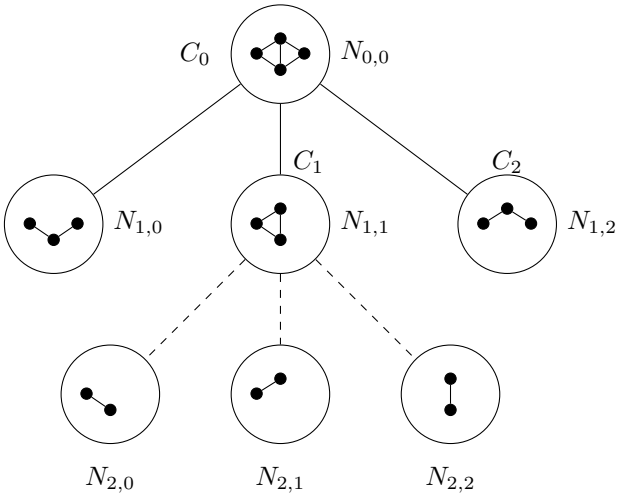

Figure 2: A possible search tree generated by the SERIAL-RB algorithm while solving the MINIMUM FEEDBACK VERTEX SET problem on G. Deleted vertices correspond to vertices that have been added to the solution and dotted lines indicate search nodes that will be pruned by the algorithm.

The goal of this paper is to transform SERIAL-RB into a scalable parallel algorithm with as little effort as possible. For ease of presentation, we make the following assumptions:

- SERIAL-RB solves an NP-hard optimization problem (i.e. minimization

or maximization) where each solution appears in a leaf of the search tree.

- The global variable $best\_so\_far$ stores the best solution found so far.
- The IsSolution($N_{d,p}$) function returns true only if $N_{d,p}$ contains a solution which is "better" than $best\_so\_far$.
- The search tree explored by Serial-RB is binary (i.e. every search node has at most two children).

In Section 4.4, we discuss how the same techniques can be easily adapted to any search tree with arbitrary branching factor. The only (minor) requirement we impose is that the number of children of a search node can be calculated on-the-fly and that generating those children (using GetNextChild($N_{d,p}$)) follows a deterministic procedure with a well-defined order. In other words, if we run Serial-RB an arbitrary number of times on the same input instance, the search-trees of all executions will be identical. The reason for this restriction will become obvious later. We note that for most graph problems, we can use integer vertex labels to guarantee identical search trees.

In a parallel environment, we denote by $\mathcal{C} = \{C_0, C_1, \ldots, C_{c-1}\}$ the set of available computing cores. The *rank* of C_i is equal to i and $|\mathcal{C}| = c$. We use the terms *worker* and *core* interchangeably to refer to some C_i participating in a parallel computation. Each search node in T corresponds to a *task*, where tasks are *exchanged* between cores using some specified encoding. We use $E(N_{d,p})$ to denote the encoding of $N_{d,p}$. When search node $N_{d,p}$ is assigned to C_i, we say $N_{d,p}$ is the *main task* of C_i. Going back to our example on Minimum Feedback Vertex Set, if $\mathcal{C} = \{C_0, C_1, C_2\}$ then one possible initial assignment of tasks to cores is shown in Figure 2. Given that search trees can easily be decomposed to subtrees, the following classical approach first comes to mind. For $\mathcal{C} = \{C_0, C_1, \ldots, C_{c-1}\}$, start by running a serial breadth-first search starting at $N_{0,0}$ until T is decomposed into c mutually exclusive subtrees T_0, T_1, ..., T_{c-1}. Then, each core C_i is assigned subtree T_i. This seemingly straightforward parallel nature of search tree decomposition is deceiving: previous work has shown that attaining scalability is far from easy [9, 10, 11, 12].

Any task in $T(N_{d,p})$ sent from C_i to some C_j, $i \neq j$, is a *subtask* for C_i and becomes the main task of C_j. The *weight of a task*, $w(N_{d,p})$, is a numerical value indicating the estimated completion time for $N_{d,p}$ relative to other tasks. That is, when $w(N_{d,p}) > w(N_{d',p'})$, we expect the exploration of $T(N_{d,p})$ to require more computational time than $T(N_{d',p'})$. Task weight plays a crucial role in the design of efficient dynamic load-balancing strategies [13, 14, 15, 16, 17]. Without any problem-specific knowledge, the "best" indicator of the weight of $N_{d,p}$ is nothing but d since estimating the size of $T(N_{d,p})$ is almost impossible. We capture this notion by setting $w(N_{d,p}) = \frac{1}{d+1}$. We say task t_1 is *heavier* (*lighter*) than task t_2 if $w(t_1) \geq w(t_2)$ ($w(t_1) < w(t_2)$). It is important to note that we use depth as an indicator of weight mainly since we want to achieve problem independence. Theoretically, this assumption is not true (in general). A simple example for Minimum Feedback Vertex Set is a p-flower graph G

with p petals, i.e. a central vertex v with p disjoint cycles starting and ending at v. Deleting v from this graph destroys all cycles and hence there exists a leaf node in T at depth one while many internal nodes of T have depth $d > 1$. Nevertheless, our experimental results show that using depth to estimate task weights performs very well in practice. We shall discuss this behavior further in Section 7.

From the standpoint of high-performance computing, practical parallel exact algorithms for hard problems mean one thing: unbounded scalability. To the best of our knowledge, the most efficient existing parallel algorithms that solve problems similar to those we consider were only able to scale to less than a few thousand (or only a few hundred) cores [12, 16, 17, 18]. One of our main motivations was to solve extremely hard instances of the Vertex Cover problem such as the 60-cell graph [19]. In earlier work, we first attempted to tackle the problem by improving the efficiency of our serial algorithm [20]. Alas, some instances remained unsolved and some required several days of execution before we could obtain a solution. The next natural step was to attempt a parallel implementation. As we encountered scalability issues, it became clear that solving such instances in an "acceptable" amount of time would require a scalable algorithm that can effectively utilize much more than the 1,024-core limit we attained in previous work [17].

We discuss the lessons we have learned and what we believe to be the main reasons for such poor scalability in Section 3. In Section 4, we present the main concepts and strategies we use to address these challenges. Finally, implementation details, experimental results, and discussions are covered in Sections 5, 6, and 7, respectively.

3. Challenges and Related Work

3.1. Communication Overhead

The most evident overhead in parallel algorithms is that of communication. Several models have already been presented in the literature including centralized (i.e. the master-worker(s) model where most of the communication and task distribution duties are assigned to a single core) [18], decentralized [15, 17], or a hybrid of both [10]. Although each model has its pros and cons, centralization rapidly becomes a bottleneck when the number of computing cores exceeds a certain threshold [18]. Even though our approach can be implemented under any communication model, we chose to follow a fully decentralized model [17].

An efficient communication model has to (i) reduce the total number of message transmissions and (ii) minimize the travel distance (number of hops) for each transmission. Unfortunately, (ii) requires detailed knowledge of the underlying network architecture and comes at the cost of portability. For (i), the message complexity is tightly coupled with the number of times each C_i runs out of work and requests more. Therefore, to minimize the number of generated messages, we need to maximize "work time", which is achieved by better dynamic load balancing.

3.2. Tasks, Buffers, and Memory Overhead

No matter what communication model is used, a certain encoding has to be selected for representing tasks in memory. A drawback of the encoding used by Finkel and Manber [21] is that every task is an exact copy of a search node, whose size can be quite large. In a graph algorithm, every search node might contain a modified version of the input graph (and some additional information). In this case, a more compact task-encoding scheme is needed to reduce both memory and communication overheads.

Almost all parallel algorithms in the literature require a task-buffer or task-queue to store multiple tasks for eventual delegation [10, 17, 18, 21]. As buffers have limited size, their usage requires the selection of a "good" parameter value for task-buffer size. Choosing the size can be a daunting task, as this parameter introduces a tradeoff between the amount of time spent on creating or sending tasks and that spent on solving tasks. It is very common for such parallel algorithms to enter a loop of multiple *lightweight task* exchanges, i.e. tasks generated near the bottom of the search tree. Such loops unnecessarily consume considerable amounts of time and memory [17] as lightweight tasks would be more efficiently solved in-place by a single core.

3.3. Initial Distribution

Efficient dynamic load balancing is key to scalable parallel algorithms. To avoid loops of multiple lightweight task exchanges, initial task distribution also plays a major role. Even with clever load-balancing techniques, such loops can consume a lot of resources and delay (or even deny) the system from reaching a balanced state.

3.4. Serial Overhead

All the items discussed above induce some serial overhead. Here we focus on encoding and decoding of tasks, which greatly affect the performance of any parallel algorithm. Upon receiving a new task, each computing core has to perform a number of operations to correctly restart the search phase, i.e. resume the exploration of its assigned subtree. When the search reaches the bottom levels of the tree, the amount of time required to start a task might exceed the time required to solve it, a situation that should be avoided. Encoding tasks and storing them in buffers also consumes time. In fact, the more we attempt to compress task encodings the more serial work is required for decoding.

For NP-hard problems, it is important to account for what we call the *butterfly effect* of polynomial overhead. Since the size of the search tree is usually exponential in the size of the input, any polynomial-time (or even constant-time) operations can have significant effects on the overall running times [20], by virtue of being executed exponentially many times. In general, the disruption time (time spent doing non-search-related work) has to be minimized.

3.5. Load Balancing

Task creation, i.e. determining when, how, and how often to create and/or distribute tasks, is one of the most critical factors affecting load balancing [18]. Careful tracing of recursive backtracking algorithms shows that most computational time is spent near the bottom of the search tree, where d is very large. Moreover, since task-buffers have fixed size, any parallel execution of a recursive backtracking algorithm relying on task-buffers is very likely to reach a state where all buffers contain lightweight tasks. Loops of multiple lightweight task exchanges most often occur in such scenarios. To avoid them, we need a mechanism that enables the extraction of a task of maximum weight from the subtree assigned to a C_i, that is, the highest unvisited node in the subtree assigned to C_i.

Several load-balancing strategies have been proposed in the literature to tackle the aforementioned problems [13, 14, 9, 12, 22]. In recent work [16], a load-balancing strategy designed specifically for the Vertex Cover problem was presented. The algorithm is based on a dynamic master-worker model where prior knowledge about generated instances is manipulated so that the core having the estimated heaviest task is selected as master. However, scalability of this approach was limited to only 2,048 cores.

3.6. Termination Detection

In a centralized model, the master detects termination using straightforward protocols. The termination protocol can be initiated several times by different cores in a decentralized environment, rendering detection more challenging. In this work, we use a protocol similar to the one proposed by Abu-Khzam et al. [17], where each core, which can be in one of three states, broadcasts any state change to all other cores.

3.7. Identifying Parallelism and Problem Independence

Another important aspect to consider in the design of parallel algorithms is the identification of parallelism in the sequential version of the algorithms. As previously noted, our focus is on NP-hard problems where, in most cases, the exploration of a single root-to-leaf path in the search tree requires time polynomial in the input size, whereas the search tree size grows exponentially. Therefore, we chose to partition the search tree of the problem and only parallelize the exploration of its different subtrees, i.e. keeping all computations executed inside a single search node serial. Another reason for this choice is that we need to address all the challenges listed above independently of the problem being solved.

4. Addressing the Challenges

In this section, we show how to incrementally transform Serial-RB into a parallel algorithm. First, we discuss indexed search trees and their use in a generic and compact task-encoding scheme. As a byproduct of this encoding,

we show how we can efficiently extract heavy (if not heaviest) unprocessed tasks for dynamic load balancing. We provide pseudocode to illustrate the simplicity of transforming serial algorithms to parallel ones. The end result is a parallel algorithm, Parallel-RB, which consists of two main procedures: Parallel-RB-Iterator and Parallel-RB-Solver.

4.1. Indexed Search Trees

For a binary search tree T, we let $left(N_{d,p})$ and $right(N_{d,p})$ denote the left child and the right child of node $N_{d,p}$, respectively. We use the following procedure to assign an index, idx, to every search node in T (where + denotes concatenation):

(1) The root of T has index 1 ($idx(N_{0,0})$ = "1")

(2) For any node $N_{d,p}$ in T:

(2.1) $idx(left(N_{d,p})) = idx(N_{d,p})$ + "0" and

(2.2) $idx(right(N_{d,p})) = idx(N_{d,p})$ + "1"

An example of an indexed binary search tree is given in Figure 3. Note that this indexing method can easily be extended for arbitrary branching factor by simply setting the index of the k^{th} child of $N_{d,p}$ to $idx(N_{d,p})$ + "$(k-1)$". The general idea of indexing is not new and has been previously used for prioritizing tasks in buffers or queues [10, 13]. However, as we shall see next, we can completely eliminate the need for buffering multiple tasks by combining a fully decentralized communication model with some operations for manipulating indices. Hence, we effectively reduce the memory footprint of our algorithms and eliminate the burden of selecting appropriate size parameters for each buffer or task granularity as described by Sun et al. [10].

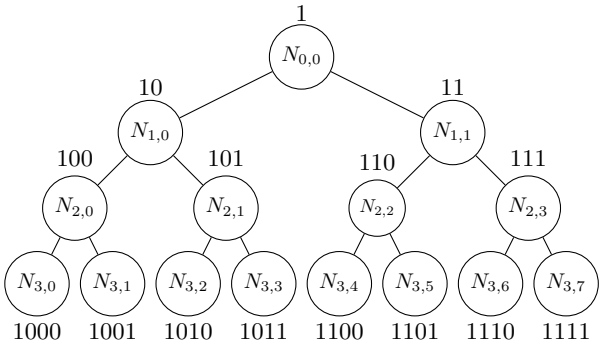

Figure 3: Example of an indexed binary search tree

To incorporate indices into our algorithm, we introduce minor modifications to Serial-RB. We call this new version Almost-Parallel-RB (Figure

4). Almost-Parallel-RB includes a global integer array *current_idx* that is maintained by a single statement: $current\_idx[d] = p$. Since we assume binary search trees, $p \in \{0, 1\}$ and whenever Almost-Parallel-RB is exploring search node $N_{d,p}$ we have the invariant that *current_idx* is an array representation of $idx(N_{d,p})$. We let $E(N_{d,p}) = idx(N_{d,p})$, i.e. the encoding of a task corresponds to its index and is of $\mathcal{O}(d)$ size. Combined with an effective load-balancing strategy which generates tasks having only small d, this approach aims at reducing memory and communication overheads. Upon receiving an encoded task, every core now requires an additional function ConvertIndex, the implementation details of which are problem-specific (Section 5 discusses some examples). The purpose of this function is to convert an index into an actual task from which the search can proceed. Since every index encodes the unique path from the root of the tree to the corresponding search node and by assumption search nodes are generated in a well-defined order, to retrace the operations it suffices to iterate over the index. Even though one is trading communication volume and memory for serial computational work, the overhead introduced by this approach remains closely related to the number of tasks solved by each core, i.e. the smaller this number the less additional serial work is required. Moreover, minimizing this number also minimizes disruption time since the search-phase of the algorithm is not affected. To do so, we introduce a mechanism allowing each core to extract its heaviest unprocessed task (or highest unvisited node in its corresponding search tree) using only the information provided by the index. We use the function GetHeaviestTaskIndex to repeatedly extract the heaviest task from the *current_idx* array and FixIndex to ensure that no search node is ever explored twice (Figure 5).

4.2. Generating the Local Heaviest Task

The GetHeaviestTaskIndex function (Figure 5) relies on the following observation:

Observation 1. *Let T be some search tree and assume C_0 is the only core exploring T using the* Almost-Parallel-RB *algorithm (Figure 4). If current_idx $= [x_0,x_1,\ldots,x_n]$, $x_i \in \{0,1\}$, then the highest unvisited node in T has index $[x_0,x_1,\ldots,x_i+1]$, where $i \leq n$ and x_i is the first zero entry in current_idx.*

Observation 1 follows from the fact that Almost-Parallel-RB explores T using depth-first search, i.e. the left child of a search node is always explored first. Hence, given our procedure for assigning indices to nodes, a zero entry in $current\_idx[d]$ indicates the existence of an unvisited "right child" at depth d (Figure 3). Searching for the first zero entry in *current_idx* guarantees that this child is the highest unvisited node in T. Of course, Observation 1 only holds for a single core. More work is needed to maintain a slightly weaker invariant in a parallel environment where cores are exploring different sections of the search tree and tasks are being exchanged. Intuitively, in a parallel environment, every core extracts the locally highest unvisited node in the subtree of T it is currently

exploring. To remember which tasks have been delegated, negative numbers are used as "markers." We discuss the details next.

Assume a parallel computation involving cores C_i and C_j and the search tree shown in Figure 3. C_i has main task $N_{0,0}$ and is currently exploring node $N_{3,2}$ (hence $current\_idx = [1, 0, 1, 0]$). After receiving an initial task request from C_j, C_i calls GetHeaviestTaskIndex. By Observation 1, for h the smallest integer such that $current\_idx[h] = 0$, $current\_idx[0 \rightarrow h]$, the subarray of $current\_idx$ starting at position 0 and ending at position h corresponds to the index of the highest unvisited node in the search tree assigned to C_i. Therefore, in our example $h = 1$ and GetHeaviestTaskIndex returns $temp\_idx = current\_idx[0 \rightarrow h] = [1, -1]$ and sets $current\_idx = [1, -1, 1, 0]$. Position h in $current\_idx$ is set to -1 to guarantee that no search node is ever explored twice. Before exploring a search node, every core must first use $current\_idx$ to validate that the current branch was not previously delegated to a different core (Figure 4, lines 2–3). Whenever C_i discovers a -1 in $current\_idx[d]$, the search can terminate, since the remaining subtree has been reassigned to a different core.

```
1:  procedure Almost-Parallel-RB(N_{d,p})
2:      if (current_idx[d] = −1) then
3:          terminate;
4:      current_idx[d] ← p;
5:      if (IsSolution(N_{d,p})) then
6:          best_so_far ← N_{d,p};
7:      if (IsLeaf(N_{d,p})) then
8:          Backtrack;                                  ▷ undo operations
9:      if (TaskRequestExists(())) then
10:         x ← GetHeaviestTaskIndex(current_idx);
11:         send(x, requester);
12:     p' ← 0;
13:     while HasNextChild(N_{d,p}) do
14:         N_{d+1,p'} ← GetNextChild(N_{d,p});
15:         Almost-Parallel-RB(N_{d+1,p'});
16:         p' ← p' + 1;
```

Figure 4: The Almost-Parallel-RB algorithm

```
1: function GetHeaviestTaskIndex(current_idx)
2:     for i ← 0, current_idx.length − 1 do
3:         if (current_idx[i] = 0) then
4:             current_idx[i] ← −1;
5:             temp_idx ← current_idx[0 → i];                ▷ subarray
6:             return temp_idx;
7:     return null;
1: function FixIndex(temp_idx)
2:     for i ← 0, temp_idx.length − 2 do
3:         if (temp_idx[i] < 0) then
4:             temp_idx[i] ← 0;
5:     temp_idx[temp_idx.length − 1] ← 1;
6:     return temp_idx;
```

Figure 5: The GetHeaviestTaskIndex and FixIndex functions

At the receiving end, C_j calls FixIndex, after which $temp\_idx = [1, 1]$. As seen in Figure 3 and by Observation 1, $N_{1,1}$ was in fact the heaviest task in $T(N_{0,0})$. C_j proceeds by converting the received index to a task and then starts exploring the corresponding subtree. If C_j subsequently requests a second task from C_i while C_i is still working on node $N_{3,2}$, the resulting task is $[1, 0, 1, 1]$ and the *current_idx* of C_i is updated to $[1, -1, 1, -1]$. When the *current_idx* of C_i contains no zero entries, C_i sends a no-task response to C_j. Both the GetHeaviestTaskIndex and FixIndex functions run in $\mathcal{O}(d)$ time.

4.3. From Serial to Parallel

The Almost-Parallel-RB algorithm is lacking a formal definition of the communication model as well as the implementation details of the initialization and termination protocols. For the former, we use a fully decentralized model in which any two cores can communicate. We assume each core is assigned a unique rank r, for $0 \leq r < c$. We consider three different types of message exchanges: status updates, task requests or responses, and notification messages. Each core can be in one of three states: active, inactive, or dead. Before changing states, each core must broadcast a status update message to all participants. This information is maintained by each core in a global integer array *statuses*. Notification messages are optional broadcast messages whose purpose is to inform the remaining participants of current progress. In our implementation, notification messages are sent whenever a new solution is found. The message includes the size of the new solution which, for many algorithms, can be used as a basis for effective pruning rules.

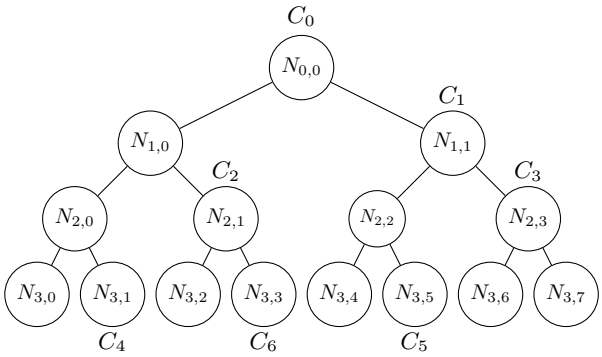

Figure 6: Example of an initial task-to-core assignment for $c = 7$

In the initialization phase, for a binary search tree and the number of cores a power of two ($c = 2^x$), one strategy would be to generate all search-nodes at depth x and assign one to each core. However, these requirements are too restrictive and greatly complicate the implementation, as search trees need not be binary and utilizing only $c = 2^x$ of out $2^{x+1} - 1$ cores would leave a lot of resources idle. Instead, we exploit the ranks of cores to arrange them in a virtual tree-like topology. Every core, except C_0, is forced to request the first task from its parent (stored as a global variable) in this virtual topology. C_0 is always assigned task $N_{0,0}$. The GetParent function, which runs in $\mathcal{O}(\log^2 c)$ time, is given in Figure 7. The intuition is that if we assume that cores join the computation in increasing order of rank, C_i must always request an initial task from C_j where $j < i$ and there exists no C_k such that $k < i$ and C_k has a heavier task than C_j. Figure 6 shows an example of an initial task-to-core assignment for $c = 7$. The parent of C_i corresponds to the first C_j encountered on the path from the task assigned to C_i to the root. When C_4 joins the topology, although all remaining cores $(C_0, \ldots, C_3)$ have tasks of equal weight, C_4 selects C_0 as a parent. This is due to the alternating behavior of the GetParent function, i.e. when i is even, the parent of C_i corresponds to C_j, where j is the smallest even integer such that C_j has a task of maximum weight. The same holds for odd i, except that C_1 must pick C_0 as a parent. This approach aims at balancing the number of cores exploring different sections of the search tree.

```
1: function GetParent(r, c)
2:     parent ← 0;
3:     for (i = 0; i < c; i++) do
4:         if (2^i > r) then
5:             break;
6:         parent ← r − 2^i;
7:     return parent;
1: function GetNextParent(r, c)
2:     parent ← (parent + 1) mod c;
3:     if (parent = r) then
4:         parent ← (parent + 1) mod c;
5:         passes ← passes + 1
6:     return parent;
```

Figure 7: The GetParent and GetNextParent functions

Once every core receives a response from its initial parent, the initialization phase is complete. After that, each core updates its parent to $(r + 1) \bmod c$. During the search-phase, task requests follow the receiver-initiated or work-stealing paradigm [12, 14] modified to fit our fully decentralized communication model. In other words, whenever a core requires a new task it will first attempt to request one from its current parent. If the parent has no available tasks or is inactive, the virtual topology is modified (in $\mathcal{O}(1)$ time) by the GetNextParent function (Figure 7).

In the global variable *passes*, we keep track of the number of times each core has unsuccessfully requested a task from all participants. The termination protocol is invoked by some core C_i whenever *passes* is incremented. C_i goes from being active to inactive and sends a status update message to inform the remaining participants. Once all cores are inactive, the computation can safely end. The complete pseudocode for Parallel-RB is given in Figure 8. The algorithm consists of two main procedures: Parallel-RB-Iterator and Parallel-RB-Solver. To minimize disruption time, all communication must be non-blocking in the latter and blocking in the former.

```
1: procedure PARALLEL-RB-ITERATOR(r, c)
2:     init ← true; passes ← 0;
3:     parent ← GETPARENT(r, c);
4:     while true do
5:         if (passes was incremented) then
6:             TERMINATIONPROTOCOL();
7:         if (r = 0 & init) then
8:             init ← false;
9:             PARALLEL-RB-SOLVER(N_{0,0});
10:        else
11:            if (init) then
12:                init ← false;
13:                idx ← REQUESTTASKINDEX(parent);
14:                parent ← (r + 1) mod c;
15:                if (idx ≠ null) then
16:                    N ← CONVERTINDEX(idx);
17:                    PARALLEL-RB-SOLVER(N);
18:        idx ← REQUESTTASKINDEX(parent);
19:        if (idx ≠ null) then
20:            N ← CONVERTINDEX(idx);
21:            PARALLEL-RB-SOLVER(N);
22:        else
23:            parent ← GETNEXTPARENT(r, c);

24: procedure PARALLEL-RB-SOLVER(N_{d,p})
25:     if (current_idx[d] = −1) then
26:         terminate;
27:     current_idx[d] ← p;
28:     if (ISSOLUTION(N_{d,p})) then
29:         best_so_far ← N_{d,p};
30:         Broadcast the new solution;                    ▷ Optional
31:     if (ISLEAF(N_{d,p})) then
32:         Backtrack;                                     ▷ undo operations
33:     if (TASKREQUESTEXISTS()) then
34:         x ← GETHEAVIESTTASKINDEX(current_idx);
35:         SEND(x, requester);
36:     if (BROADCASTMESSAGEEXISTS()) then
37:         Read and perform necessary actions;
38:     p' ← 0;
39:     while HASNEXTCHILD(N_{d,p}) do
40:         N_{d+1,p'} ← GETNEXTCHILD(N_{d,p});
41:         PARALLEL-RB-SOLVER(N_{d+1,p'});
42:         p' ← p' + 1;
```

Figure 8: The PARALLEL-RB algorithm

4.4. Arbitrary Branching Factor

For search trees of arbitrary branching factor, the index of $N_{d,p}$ needs to keep track of both the unique root-to-node path as well as the number of unexplored siblings of $N_{d,p}$ (i.e. all the nodes at depth d and position greater than p). Therefore, we divide an index into two parts, idx_1 and idx_2. We let $k^{th}(N_{d,p})$ denote the k^{th} child of $N_{d,p}$ and $C(N_{d,p})$ the set of all children of $N_{d,p}$. The following procedure assigns indices to every search-node in T:

(1) The root of T has $idx_1(N_{0,0})$ = "1" and $idx_2(N_{0,0})$ = "0"

(2) For any node $N_{d,p}$ in T:

(2.1) $idx_1(k^{th}(N_{d,p})) = idx_1(N_{d,p}) +$ "$(k-1)$" and

(2.2) $idx_2(k^{th}(N_{d,p})) = idx_2(N_{d,p}) +$ "$(|C(N_{d,p})| - k)$"

An example of an indexed search tree is given in Figure 9. Each node is assigned two identifiers: idx_1 (top) and idx_2 (bottom). At the implementation level, the *current_idx* array is replaced by a $2 \times d$ array that can be maintained after every recursive call in a fashion similar to Parallel-RB-Solver as long as each search-node $N_{d,p}$ is aware of $|C(N_{d,p})|$.

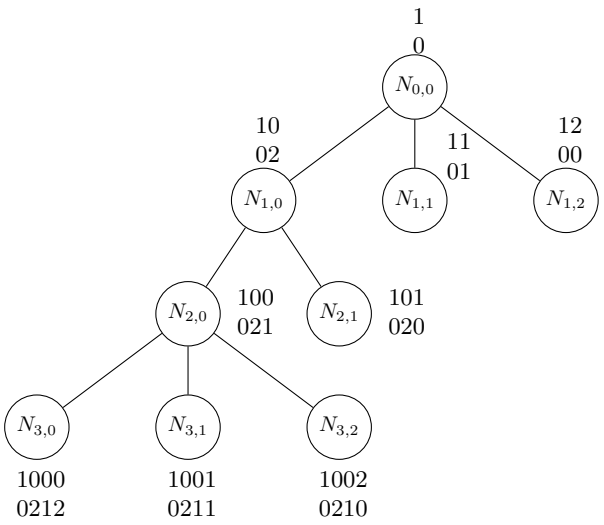

Figure 9: Example of an indexed search tree with arbitrary branching factor

The first non-zero entry in *current_idx*[1] (the second row of the array), say *current_idx*[1][x], indicates the depth of all tasks of heaviest weight. Since there can be more than one unvisited node at this depth, we could choose to send either all of them or just a subset S. In the first case, we can remember

delegated branches by simply setting $current\_idx[1][x]$ to -1. For the second case, $current\_idx[1][x]$ is decremented by $|S|$. Note that the choice of S cannot be arbitrary. If $C(N_{d,p}) = \{N_{d+1,0}, N_{d+1,1}, \ldots, N_{d+1,p_{max}}\}$, S must include $N_{d+1,p_{max}}$, and for any $N_{d+1,i} \in S$, it must be the case that $N_{d+1,j}$ is also in S for all j between i and p_{max}. The only modification required in Parallel-RB-Solver is to make sure that at search-node $N_{x,p}$, GetNextChild is executed only $current\_idx[1][x]$ times.

5. Implementation

We tested our approach with parallel implementations of algorithms for the well-known Vertex Cover and Dominating Set problems.

Vertex Cover
Input: A graph $G = (V, E)$
Question: Find a set $C \subseteq V$ such that $|C|$ is minimized and the graph induced by $V \setminus C$ is edgeless

Dominating Set
Input: A graph $G = (V, E)$
Question: Find a set $D \subseteq V$ such that $|D|$ is minimized and every vertex in G is either in D or is adjacent to a vertex in D

Both problems have received considerable attention in the areas of exact and fixed parameter algorithms because of their close relations to many other problems in different application domains [23]. The sequential algorithm for the parameterized version of Vertex Cover having the fastest known worst-case behavior runs in $\mathcal{O}(kn + 1.2738^k)$ time [3], where k is an upper bound on the size of the target solution. We converted this to an optimization version by introducing simple modifications and excluding complex processing rules that require heavy maintenance operations. For Dominating Set, we implemented the algorithm of Fomin et al. [4] where the problem is solved by a reduction to Minimum Set Cover. We used the hybrid graph data structure [20], specifically designed for recursive backtracking algorithms, that combines the advantages of the two classical adjacency-list and adjacency-matrix representations of graphs with very efficient implicit backtracking operations.

Our input consists of a graph $G = (V, E)$ where $|V| = n$, $|E| = m$, and each vertex is given an identifier between 0 and $n - 1$. The search tree for each algorithm is binary and the actual implementations closely follow the Parallel-RB algorithm. At every search-node, a vertex v of highest degree is selected deterministically. Vertex selection has to be deterministic to meet the requirements of our approach. To break ties when multiple vertices have the same degree, we always pick the vertex with the smallest identifier. For Vertex Cover, the left branch adds v to the solution and the right branch adds all the neighbors

of v to the solution. For Dominating Set, the left branch is identical but the right branch forces v to be out of any solution. The ConvertIndex function is straightforward as the added, deleted, or discarded vertices can be retraced by iterating through the index and applying any appropriate reduction rules along the way. Every time a smaller solution is found, the size is broadcasted to all participants to avoid exploring branches that cannot lead to any improvements.

6. Experimental Results

Our code, written in the standard C language, utilizes the Message Passing Interface (MPI) [24] and has no other dependencies. Computations were performed on the BGQ supercomputer at the SciNet HPC Consortium[2]. The BGQ production system is a 3^{rd} generation Blue Gene IBM supercomputer built around a system-on-a-chip compute node. There are 2,048 compute nodes each having a 16 core 1.6GHz PowerPC based CPU with 16GB of RAM. When running jobs on 32,768 cores, each core is allocated 1GB of RAM. Each core also has four "hardware threads" that can keep the different parts of each core busy at the same time. It is therefore possible to run jobs on $65,536$ and $131,072$ cores at the cost of reducing available RAM per core to 500MB and 250MB, respectively. We could run experiments using this many cores only when the input graph was relatively small and, due to the fact that multiple cores were forced to share (memory and CPU) resources, we noticed a slight decrease in performance.

The Parallel-Vertex-Cover algorithm was tested on four input graphs.

- p_hat700-1.clq: 700 vertices and $234,234$ edges with a minimum vertex cover of size 635
- p_hat1000-2.clq: $1,000$ vertices and $244,799$ edges with a minimum vertex cover of size 946
- frb30-15-1.mis: 450 vertices and $17,827$ edges with a minimum vertex cover of size 420
- 60-cell: 300 vertices and 600 edges with a minimum vertex cover of size 190

The first two instances were obtained from the classical Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) benchmark suite (`http://dimacs.rutgers.edu/Challenges/`). The frb30-15-1.mis graph is a notoriously hard instance for which the exact size of a solution was known so far only from a theoretical perspective. To the best of our knowledge, this paper is the first to experimentally solve it; more information on this instance can be

[2]SciNet is funded by the Canada Foundation for Innovation under the auspices of Compute Canada; the Government of Ontario; Ontario Research Fund - Research Excellence; and the University of Toronto. [25]

Table 1: Parallel-Vertex-Cover statistics on graphs p_hat700-1.clq and p_hat1000-2.clq.

Graph	|C|	Time	T_S	T_R	Speedup
p_hat700-1.clq	16	19.5hrs	2,876	2,910	
p_hat700-1.clq	32	9.8hrs	2,502	2,567	1.99
p_hat700-1.clq	64	4.9hrs	3,398	3,518	2.00
p_hat700-1.clq	128	2.5hrs	4,928	5,196	1.96
p_hat700-1.clq	256	1.3hrs	4,578	5,153	1.92
p_hat700-1.clq	512	38min	4,354	5,451	2.05
p_hat700-1.clq	1,024	18.9min	4,052	6,391	2.01
p_hat700-1.clq	2,048	9.89min	3,781	8,117	1.91
p_hat700-1.clq	4,096	5.39min	3,665	11,978	1.83
p_hat700-1.clq	8,192	2.9min	2,714	19,183	1.86
p_hat700-1.clq	16,384	1.7min	1,342	32,883	1.71
p_hat1000-2.clq	64	23.6min	3,664	3,799	
p_hat1000-2.clq	128	12.5min	2,651	2,912	1.89
p_hat1000-2.clq	256	6.5min	1,623	1,956	1.92
p_hat1000-2.clq	512	3.7min	1,235	1,872	1.75
p_hat1000-2.clq	1,024	2.1min	866	2,142	1.76
p_hat1000-2.clq	2,048	1.2min	610	3,120	1.75

found in the work of Xu et al. [26]. Lastly, the 60-cell graph is a 4-regular graph (every vertex has exactly 4 neighbors) with applications in chemistry [19]. Prior to this work, we solved the 60-cell using a serial algorithm which ran for almost a full week [20]. The high regularity of the graph makes it very hard to apply any pruning rules, resulting in an almost exhaustive enumeration of all feasible solutions. For the Parallel-Dominating-Set algorithm we generated two random instances 201x1500.ds and 251x6000.ds where nxm.ds denotes a graph on n vertices and m edges. Neither instance could be solved by our serial algorithm when limited to 24 hours.

All of our experiments were limited by the system to a maximum of 24 hours per job. To evaluate the performance of our communication model and dynamic load balancing strategy, we collected two statistics from each run: T_S and T_R. T_S denotes the average number of tasks received (and hence solved) by each core while T_R denotes the average number of tasks requested by each core. In Tables 1 and 2, we give the running times of the Parallel-Vertex-Cover algorithm for every instance while varying the total number of cores, $|C|$, from 2 to $131,072$ (we only ran experiments on 65,536 or 131,072 cores when the graph was small enough to fit in memory or when the running time exceeded

Table 2: Parallel-Vertex-Cover statistics on graphs frb30-15-1.mis and 60-cell.

Graph	|C|	Time	T_S	T_R	Speedup
frb30-15-1.mis	1,024	14.2hrs	13,580	15,968	
frb30-15-1.mis	2,048	7.2hrs	21,899	26,597	1.97
frb30-15-1.mis	4,096	3.6hrs	28,740	37,733	2.01
frb30-15-1.mis	8,192	1.9hrs	29,110	45,685	1.89
frb30-15-1.mis	16,384	55.1min	28,707	59,978	2.07
frb30-15-1.mis	32,768	28.8min	30,008	96,438	1.91
frb30-15-1.mis	65,536	16.8min	25,359	158,371	1.71
frb30-15-1.mis	131,072	11.1min	19,419	312,430	1.51
60-cell	128	14.3hrs	19	26	
60-cell	256	7.3hrs	23	23	1.96
60-cell	512	3.7hrs	1,091	1,388	1.97
60-cell	1,024	45.1min	1,397	1,940	4.92
60-cell	2,048	11.3min	1,331	2,430	3.99
60-cell	4,096	2.8min	949	3,094	4.04

10 minutes on 32,768 cores).

The values of T_S and T_R are also provided. Since we double the number of cores after every run, speedup values for each row are based on the running time recorded for the previous row. Similar results for the Parallel-Dominating-Set algorithm are given in Table 3. We show the overall behaviors in the chart of Figure 10.

In Figure 11, we plot the different values of T_S and T_R for a representative subset of our experiments. This chart reveals the inherent difficulty of dynamic load balancing. As $|C|$ increases, the gap between T_S and T_R widens. We believe that any efficient dynamic load-balancing strategy has to control the growth of this gap (e.g. keep it linear) for a chance to achieve unbounded scalability. The largest gap we obtained was approximately 300,000 on the frb30-15-1.mis instance using 131,072 cores. Given the number of cores and the amount of time ($>$ 10 minutes) spent on the computation, the number suggests that each core requested an average of 2.5 tasks from every other core. One possible improvement which we are currently investigating is to modify our virtual topology to a graph-like structure of bounded degree. Our approach currently assumes a fully connected topology after initialization (i.e. any two cores can communicate) which explains the correlation between $|C|$ and $|T_S - T_R|$. By bounding the degrees in the virtual topology, we hope to make this gap weakly dependent on $|C|$.

Table 3: Parallel-Dominating-Set statistics on random graphs.

Graph	|C|	Time	T_S	T_R	Speedup
201x1500.ds	512	18.1hrs	8,231	9,642	
201x1500.ds	1,024	9.2hrs	10,315	12,611	1.97
201x1500.ds	2,048	4.5hrs	11,566	16,118	2.04
201x1500.ds	4,096	2.3hrs	14,070	23,413	1.96
201x1500.ds	8,192	1.2hrs	13,243	33,680	1.92
201x1500.ds	16,384	36.2min	10,295	41,795	1.99
201x1500.ds	32,768	19.2min	6,925	72,719	1.89
201x1500.ds	65,536	11.8min	4,221	109,346	1.63
251x6000.ds	256	8.9hrs	3,313	4,573	
251x6000.ds	512	4.7hrs	3,865	4,985	1.89
251x6000.ds	1,024	2.4hrs	2,842	5,306	1.96
251x6000.ds	2,048	1.2hrs	1,528	5,396	2.00
251x6000.ds	4,096	36.4min	2,037	9,714	1.98
251x6000.ds	8,192	18.7min	1,445	10,497	1.95
251x6000.ds	16,384	10.1min	1,132	12,310	1.85
251x6000.ds	32,768	5.5min	934	13,982	1.84

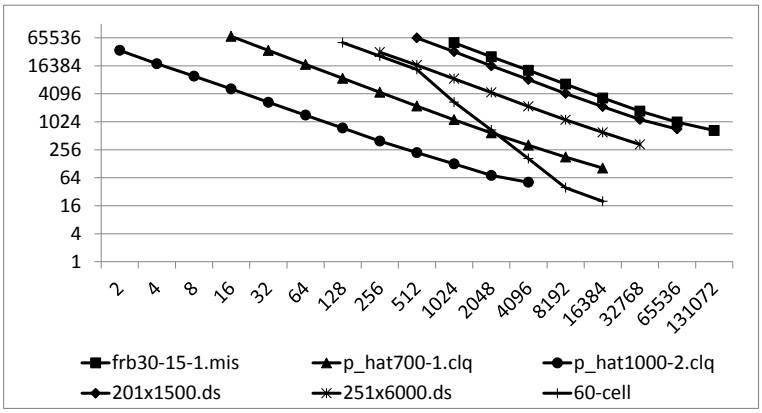

Figure 10: The logarithm (base 2) of running times in seconds (y-axis) vs. number of cores (x-axis)

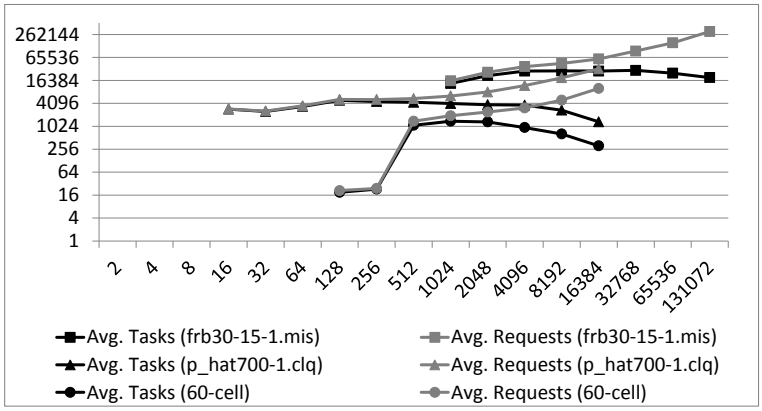

Figure 11: The logarithm (base 2) of the average number of message transmissions (y-axis) vs. number of cores (x-axis) (T_S shown in black and T_R shown in gray)

7. Interpreting the Results

In almost all cases, the algorithms achieve near linear speedup on at least 32,768 cores. Not surprisingly, whenever the time required to solve an instance drops to just a few minutes, the overall performance of the algorithms decreases as we add more cores to the computation. More surprising might be the super-linear speedups attained for the 60-graph. This is mainly due to some sort of cooperation among the various cores: when one core finds a solution of a certain "improved" size, the value of $best\_so\_far$ gets updated. Consequently, some cores might discover that a better solution cannot be obtained in their respective subtrees, hence allowing for pruning of potentially large sections of the search tree. This behavior is highly effective when the number of optimum solutions is relatively small (very few leaf-nodes in the search-tree correspond to optimum solutions).

From the results of 201x1500.ds and frb30-15-1.mis on 65,536 cores, we expected a performance slowdown of about 10 percent in general, as running on more cores would force sharing of resources whenever $|C|$ is greater than 32,768 (on the BGQ system). Since all of the problem instances we tested on were solved in just a few minutes using at most 32,768 cores, we hope that "appropriately" harder instances will be available in the future to fairly test scalability on a larger number of cores.

Although we use depth as an indicator of task weight and restrict our attention to locally highest unvisited nodes, experimental results show promising scalability for a very large number of cores. We believe that this is due to the simplicity of our approach and the dynamic nature of the virtual topology; as the computation progresses, clusters of cores exploring some section of the search space are formed. Even though some cores in other clusters could have locally heavier tasks, every cluster is "efficiently" exhausting its search space and then "breaking up" to join the remaining "busier" clusters. Here, efficiency

is very likely tied to the fact that the indexing approach greatly reduces the amount of time spent on parallelization and focuses on keeping every core busy exploring the search space.

8. Conclusions and Future Work

Combining indexed search trees and (local) heaviest task extraction with a decentralized communication model, we have showed how any serial recursive backtracking algorithm, with some ordered branching, can be modified to run in parallel. Some of the key advantages of our approach are:

- The migration from serial to parallel entails very little additional coding. Implementing each of our parallel algorithms took less than two days.
- It completely eliminates the need for buffering multiple tasks and the overhead they introduce (Section 3.2).
- The inputs of the serial and parallel implementations are identical. Running the parallel algorithms requires no additional input from the user (assuming every core has access to r and c). Most parallel algorithms in the literature require some parameters such as task-buffer size. Selecting the best parameters could vary depending on the instance being solved.
- Experimental results have showed that our implicit load-balancing strategy, joined with the concise task-encoding scheme, can achieve nearly linear (sometimes super-linear) speedup with scalability on at least $32,768$ cores. We hope to test our implementations on a larger system in the near future to determine the maximum number of cores it can support.
- Although not typical of parallel algorithms, when using the indexing method and the ConvertIndex function, it becomes reasonably straightforward to support join-leave (i.e. cores leaving the computation after solving a fixed number of tasks) or checkpointing capabilities (i.e. by forcing every core to write its *current_idx* to disk).

We believe there is still plenty of room for improvements at the risk of losing some of the simplicity. One area we have started examining is the virtual topology. A "smarter" topology could further reduce the communication overhead (e.g. the gap between T_S and T_R) and increase the overall performance. One possibility is to adapt the randomized work-stealing approach to a fully decentralized communication model [14, 12]. Another candidate is the GetNextParent function which can be modified to probe a fixed number of cores before selecting which to "help" next. Finally, we intend to investigate the possibility of developing our approach into a framework or library, similar to previous work [10, 11], which will provide users with built-in functions for parallelizing recursive backtracking algorithms.

Acknowledgment

The authors would like to thank Chris Loken and the SciNet team for providing access to the BGQ production system and for their support throughout the experiments.

References

[1] J. Chen, I. A. Kanj, W. Jia, Vertex cover: further observations and further improvements, Journal of Algorithms 41 (2) (2001) 280–301.

[2] F. V. Fomin, D. Kratsch, G. J. Woeginger, Exact (exponential) algorithms for the dominating set problem, in: Graph-Theoretic Concepts in Computer Science, Vol. 3353, 2005, pp. 245–256.

[3] J. Chen, I. A. Kanj, G. Xia, Improved upper bounds for vertex cover, Theor. Comput. Sci. 411 (40-42) (2010) 3736–3756.

[4] F. V. Fomin, F. Grandoni, D. Kratsch, Measure and conquer: Domination a case study, in: Automata, Languages and Programming, Vol. 3580, 2005, pp. 191–203.

[5] J. M. M. Rooij, J. Nederlof, T. C. Dijk, Inclusion/exclusion meets measure and conquer, in: Algorithms - ESA 2009, Vol. 5757, 2009, pp. 554–565.

[6] G. J. Woeginger, Exact algorithms for NP-hard problems: a survey, in: Combinatorial optimization - Eureka, you shrink!, 2003, pp. 185–207.

[7] R. G. Downey, M. R. Fellows, Parameterized complexity, Springer-Verlag, New York, 1997.

[8] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.

[9] L. V. Kale, Comparing the performance of two dynamic load distribution methods, in: Proceedings of the 1988 International Conference on Parallel Processing, 1988, pp. 8–11.

[10] Y. Sun, G. Zheng, P. Jetley, L. V. Kale, An adaptive framework for large-scale state space search, in: Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum, 2011, pp. 1798–1805.

[11] T. K. Ralphs, L. Ládanyi, M. J. Saltzman, A library hierarchy for implementing scalable parallel search algorithms, J. Supercomput. 28 (2) (2004) 215–234.

[12] P. Sanders, Massively parallel search for transition-tables of polyautomata, in: Parcella 94, VI. International Workshop on Parallel Proccessing by Cellular Automata and Arrays, 1994, pp. 99–108.

[13] W. F. Clocksin, H. Alshawi, A method for efficiently executing horn clause programs using multiple processors, New Gen. Comput. 5 (4) (1988) 361–376. doi:10.1007/BF03037415.
URL http://dx.doi.org/10.1007/BF03037415

[14] G. Karypis, V. Kumar, Unstructured tree search on SIMD parallel computers, IEEE Transactions on Parallel and Distributed Systems 5 (10) (1994) 1057–1072.

[15] G. D. Fatta, M. R. Berthold, Decentralized load balancing for highly irregular search problems, Microprocess. Microsyst. 31 (4) (2007) 273–281.

[16] D. P. Weerapurage, J. D. Eblen, G. L. Rogers, Jr., M. A. Langston, Parallel vertex cover: a case study in dynamic load balancing, in: Proceedings of the Ninth Australasian Symposium on Parallel and Distributed Computing, Vol. 118, 2011, pp. 25–32.

[17] F. N. Abu-Khzam, A. E. Mouawad, A decentralized load balancing approach for parallel search-tree optimization, in: Proceedings of the 2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies, 2012, pp. 173–178. doi:10.1109/PDCAT.2012.16.

[18] F. N. Abu-Khzam, M. A. Rizk, D. A. Abdallah, 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.

[19] S. Debroni, E. Delisle, W. Myrvold, A. Sethi, J. Whitney, J. Woodcock, P. W. Fowler, B. de La Vaissiere, M. Deza, Maximum independent sets of the 120-cell and other regular polyhedral, Ars Mathematica Contemporanea 6 (2) (2013) 197–210.

[20] F. N. Abu-Khzam, M. A. Langston, A. E. Mouawad, C. P. Nolan, A hybrid graph representation for recursive backtracking algorithms, in: Proceedings of the 4th international conference on Frontiers in algorithmics, 2010, pp. 136–147.

[21] R. Finkel, U. Manber, DIB - a distributed implementation of backtracking, ACM Trans. Program. Lang. Syst. 9 (2) (1987) 235–256.

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

[23] F. N. Abu-Khzam, M. A. Langston, P. Shanbhag, C. T. Symons, Scalable parallel algorithms for FPT problems, Algorithmica 45 (3) (2006) 269–284.

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

[25] C. Loken, D. Gruner, L. Groer, R. Peltier, N. Bunn, M. Craig, T. Henriques, J. Dempsey, C.-H. Yu, J. Chen, L. J. Dursi, J. Chong, S. Northrup, J. Pinto, N. Knecht, R. V. Zon, Scinet: Lessons learned from building a power-efficient top-20 system and data centre, Journal of Physics: Conference Series 256 (1) (2010) 012026.

[26] K. Xu, F. Boussemart, F. Hemery, C. Lecoutre, A simple model to generate hard satisfiable instances, in: Proceedings of the 19th international joint conference on Artificial intelligence, 2005, pp. 337–342.