Scalable Distributed Depth-First Search with Greedy Work Stealing

Joxan Jaffar Andrew E. Santosa Roland H.C. Yap Kenny Q. Zhu
School of Computing
National University of Singapore
Republic of Singapore

E-mail: {joxan,andrews,ryap,kzhu}@comp.nus.edu.sg

Abstract

We present a framework for the parallelization of depth-first combinatorial search algorithms on a network of computers. Our architecture is intended for a distributed setting and uses a work stealing strategy coupled with a small number of primitives for the processors (which we call workers) to obtain new work and to communicate to other workers. These primitives are a minimal imposition and integrate easily with constraint programming systems. The main contribution is an adaptive architecture which allows workers to incrementally join and leave and has good scaling properties as the number of workers increases. Our empirical results illustrate that near-linear speedup for backtrack search is achieved for up to 61 workers. It suggests that near-linear speedup is possible with even more workers. The experiments also demonstrate where departures from linearity can occur for small problems, and also for problems where the parallelism can itself affect the search as in branch and bound.

1. Introduction

Search problems, such as those from constraint and combinatorial problems, involve large amounts of computation and are natural candidates for parallelization. Most of the work in parallel search [5] assumes a parallel computer with a fixed number of CPUs. Karp and Zhang [8] have shown that linear speedup for backtrack enumeration search is theoretically possible using an idealized PRAM model with a fixed number of processors. The Cilk programming model [3] shows that for multi-threaded computation on shared memory parallel computers, greedy randomized work stealing can theoretically achieve near linear speedup. Similar randomized stealing strategies are also implemented in many other systems. An important drawback for much of the theoretical work is that the idealized assumptions do not lead to scalable systems.

Today, the most cost effective and largest available computing resources are usually with networked computing, e.g. a Beowulf cluster consists of a collection of workstation/PC class machines connected using a fast network. Distributed computation running across the Internet, e.g. SETI@home or `distributed.net`, can potentially achieve peak parallelism of tens of thousands of machines. We call this inexpensive approach to parallel computing “commodity parallelism”. Among the top 500 supercomputer sites in the world, many of the new entrants are simply large PC clusters.

Early work on distributed search includes the Distributed Implementation of Backtracking (DIB) in [4], which presented a message-passing based distributed backtracking implementation. Stack splitting, receiver-initiated load balancing and a redundancy-based fault tolerance were adopted in the DIB. The method was distributed but the experimental results show saturation in execution time with just 16 CPU’s. Along such lines, a number of other systems have been built, including Atlas [1], Satin [12], Javelin [9], etc. These systems are high-level distributed programming environment suitable for wide-area networks or grids. Many also inherit the *spawn/sync* divide-and-conquer programming style from Cilk.

There is also a wealth of empirical work using parallel logic programming and constraint programming systems. OR-parallel logic programming gives a language approach for parallel search. In the distributed setting, PALS [13] is an OR-parallel logic programming system for Beowulf clusters. A comprehensive survey of parallel logic programming including OR-parallelism can be found in [6]. [11] gives empirical results for a distributed search engine in Oz but only on a 3 node cluster. A significant difference with our paper is that in much of the parallel and distributed work in logic and constraint programming, the important issue of scalability to large numbers of compute nodes is not ad-

dressed nor is there analysis of the speedup.

Our goal is to exploit the cheap compute resources of commodity parallel hardware for depth first based constraint search. This necessitates a distributed collection of computers which we call *workers* connected by a network. The number of available workers is meant to be dynamic (growing/shrinking over time), e.g. sharing a cluster among a few parallel compute jobs. We want a solution which can scale to potentially (very) large number of workers and thus avoid maintaining global information about all workers. This is in contrast with the earlier randomized load balancing schemes for parallel programming which use a centralized repository to store and manage all workers. Furthermore, because of the distributed dynamic worker setting, we want solutions which only need limited connectivity between the workers.

It is difficult to be able to obtain linear speedup for distributed search in the general case as the number of workers increases. We show rather that with our architecture, there is a region of near linear speedup as long as the amount of work in the problem is very much larger than the number of workers together with the communication costs. As the number of workers further increases, the overhead of distribution becomes too high for the problem and can lead to slowdown. This result seems reasonable given the distributed nature of the workers. Our preliminary experimental results support this analysis and show that using 61 workers in a 64 node cluster, good linear speedup is obtained. We also show that scalability is reduced for optimization problems with branch and bound search since load balancing is more difficult and overheads increase. In summary, our search architecture gives a scalable, language neutral, constraint-based search architecture which can achieve good theoretical and practical speedup on cost effective parallelism computing such as networked clusters.

2. Constraints and search

A (combinatorial) search problem P can be expressed in a general form as a 4-tuple: (V, D, C, O) where $\mathcal{V}$ is a set of variables over the domain $\mathcal{D}$, $\mathcal{C}$ is a set of constraints which are relations over those variables, and O is an optional objective function defined on $\mathcal{V}$ (in case P is an optimization problem). A *solution* of P is a valuation for $\mathcal{V}$ which satisfy the constraints $\mathcal{C}$. A special case is an optimization problem where the solution has the additional property that the value of the objective function is minimal (or maximal). The search problem is to find either some or all solutions of P.

Consider a constraint store C, the solution space can be split in two by posting two constraints, c_1 and $\neg c_1$, which creates two new subproblems, $c \wedge c_1$ and $c \wedge \neg c_1$ (see figure 1). Each subproblem is independently solvable and more specific than the original because of the added constraints. The subproblems can be further split recursively until the problem is solved. A problem is solved when the desired valuation is found or we determine that C is unsatisfiable. In general the splitting could be k-way, without loss of generality, we will only consider binary splitting here. The splitting process can be simply represented as a binary constraint search tree where the nodes represent the constraint store and the edges identify the splitting constraint.

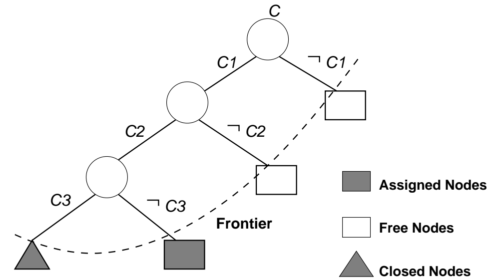

Figure 1. Frontier of constraint search tree

One can see that the constraint search tree is a generalized representation of the search space. In general, the method to choose the specific constraints $c_1, c_2, \cdots$, is left unspecified. In our distributed search, just as in sequential depth-first based search, the splitting choices can be generated dynamically which means the search tree is determined at runtime. Note that the shape of the search tree will be different with different choices. The typical search heuristics used for sequential constraint programs can be used. During distributed search, workers can even use different solvers, choices and heuristics.

The search progress is characterized by the *current search tree*, which is the set of nodes that have been explored so far. The leaf nodes of the current search tree, collectively known as the *frontier*, consists of *closed nodes*, *assigned nodes*, and *free nodes*. Closed nodes correspond to subproblems that have been solved. Assigned nodes correspond to subproblems that have been assigned to some computing agent and is currently being actively worked on, hence they are the "live" nodes. Free nodes correspond to subproblems that are not assigned to any workers. The frontier covers the whole search problem, which means when all the frontier nodes are solved, the whole problem is also solved. The frontier can *grow*, when an assigned node is split to become a new assigned node and a free node. It also *shrinks* when all the child nodes of a parent node are closed, and the parent nodes becomes the closed node.

2.1. Search engines

Our framework supports depth first based search engines with the following program structure:

1. `steal` a subproblem c – a subproblem can be a subtree of the assigned node of a given worker, which would be a *local steal*, or it could have finished searching its assigned node and needs to steal c from another worker, which would be a *global steal*.
2. if c is sufficiently small, then directly solve c sequentially;
3. otherwise, make a splitting constraint c_1, `spawn` the new subproblem $c \wedge c_1$ (which then becomes a candidate for stealing by this or another worker) and continue working on the other subproblem $c \wedge \neg c_1$.
4. goto 1.

This framework has a very close correspondence to a sequential depth first search. Consider a backtracking depth first based search engine. Splitting the problem is simply pushing the alternative choice on the stack – this corresponds to a `spawn`. When a subproblem is finished, backtracking happens, which pops the stack – this corresponds to a local steal. So if there is only one worker, the distributed search can be reduced to a backtracking one. However, non-backtracking implementations such as constraint store copying together with recomputation are also possible [10]. The main difference with sequential search is that distributed search with multiple workers will also have global steals which will reduce the number of subproblems to be stolen locally.

There are two more important operations: `publish` and `done`. The `publish` operation is used in optimization to distribute new bounds to all the workers. The `done` operation informs the system that a solution is found and optionally to terminate the caller. It should be obvious that this scheme integrates very easily with most constraint programming systems

3. Architecture

The central element of the distributed search is an architecture for managing the workers. It consists of a distinguished *master* and a varying number of *workers* and a *work-stealing* algorithm. We assume that the masters and workers are connected by a network but do not share any memory. The master does not employ any centralized data structures to maintain a list of *all* the workers. So the framework is completely distributed.

The master serves as the entry point for a worker to join a computation. It has a known address so that any potential worker can contact it. We assume that apart from an initialization phase, the master is not a bottleneck because multiple global steals do not happen every often. The master maintains a queue to serialize the requests. We assume that workers and the master share a common constraint format for receiving the subproblems. However, it is not necessary for workers to be running the same search program as long as it is a correct depth first based one using the framework for the common problem.

The master maintains the search problem definition – the root node c of the search tree. As workers join the computation, they are connected through the master to form a *worker tree*. The master also records the solutions found, propagates bounds for optimization problems and terminates the entire computation.

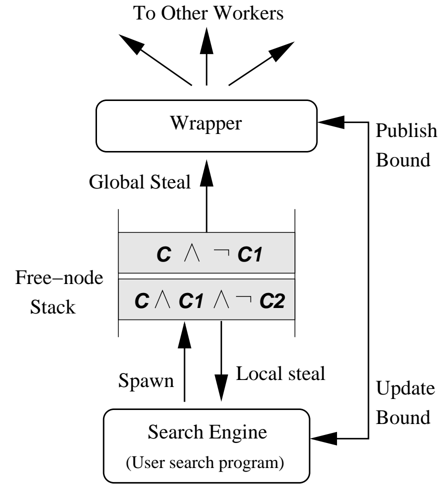

Figure 2. Structure of the worker

A worker which joins a search problem performs three operations: exploring the search tree; sharing work with other workers; and updating the system with solutions and new bounds for the objective function when they are found. A worker is divided into two processes or threads: a *wrapper* and a *search engine* (Fig. 2). The wrapper is the component of the framework responsible for dealing with work stealing and communication with the master and other workers while the search engine is the user search program which uses the framework's primitives. Interaction between the wrapper and search engine is meant to be small but does constitute overhead.

Each worker manages its remaining subproblems with a *freenode stack*. New subproblems are generated using some depth first computation on this worker. A spawn operation pushes a freenode (representing a new subproblem which has not been searched) on the stack. When a worker steals, and its *own* stack is nonempty, it removes the *most recently generated* subproblem from the stack. As this operation is

local to the worker, it encounters little overhead compared to global communication costs. We discuss later what happens if a steal returns a subproblem from another worker.

Since the master records the final solution to the original problem, the workers must report their (possibly non-final) solutions to the master from time to time. Depending on whether it is a single-solution, all-solution or best-solution mode, the master stores the solutions accordingly.

To prevent the master or any worker from being a communication bottleneck, and to allow for the coordination of a potentially large number of workers, we arrange the master and workers in a balanced worker tree where the weight on an edge records the number of nodes in the subtree from that edge. In this paper, we assume without loss of generality a binary worker tree. Initially, the worker tree has only one master node as its root, and the two edges going out of the root have zero weight. A worker is inserted into the tree by following an edge with smaller (or equal) weight unless it arrives at a node with a zero weight edge. The new worker is then inserted at that edge. During the insertion process, the weights of the affected edges are incremented by one accordingly. Traversal down the tree involves message passing between a parent node and one of its children. The resulting worker tree is a *perfectly weight-balanced tree*, where the weight of left subtree and the weight of right subtree differ by at most one. The height of such a worker tree of p workers is bounded by $\lceil \log(p) \rceil$. Notice that a balanced worker tree is constructed purely by using message passing and local worker operations. When the number of workers becomes fixed, this tree becomes static, and no further insertion cost will be incurred.

To report a new bound, a worker uses the `publish` primitive. The master will receive the solution in $O(\log(p))$ time after the message propagates up in the worker tree. Some sequential search algorithms such as branch-and-bound expedite the search process by introducing bounds and constantly updating them so that parts of the search tree can be pruned off. We have thus made `publish` a broadcast command. This is implemented by a *flooding* mechanism. A worker who receives a bound from one of its channels will record the bound and forward it to all other channels only if the bound is better than its local bound. This way, a publish of a bound takes $O(\log(p))$ time to reach every worker in the system. The updated bound is available to the search engine when a spawn or a *local steal* occurs.

3.1. Global work stealing algorithm

We now introduce a global work stealing algorithm which works using the balanced worker tree. The global steal strategy is to use a heuristic to steal the largest amount of work – this is intended to ensure good work balancing between the workers. The master first forwards the request down the worker tree to all workers. Each worker estimates the amount of work which would be available if its oldest subproblem is stolen. A simple estimate is the height of the search tree represented by that subproblem. For example, this could be estimated as the number of non-ground variables. A more sophisticated strategy would factor in the cardinality of the variable domains. The workers at the leaves of the worker tree report their address and work estimate back to their parents. An internal worker node chooses between the work estimates of itself and its children and propagates this up.

Eventually the master gets the best estimate with the address of the worker. It then instructs the "thief" to steal the top freenode directly from that worker. The total global steal time is then $O(\log(p))$. In contrast, shared-memory architecture such as Cilk can achieve constant time global stealing, at the expense of scalability.

We can show the following result for the search time T_p for p workers using the global work stealing algorithm.

Theorem 1 *Consider the execution of the greedy global steal algorithm for parallel depth-first search on arbitrary search tree with work* T_1*, height* h*, and message passing cost* u *per message. The execution time on* p *workers is* $T_p = O(T_1/p + uhp \log p)$.

The details of the analysis are omitted for lack of space.

4. Experimental results

We present some preliminary experimental results on a prototype implementation of our search framework. The experiments were run on a 64-node cluster of PCs running Linux. Each node of the cluster consists of two 1.4GHz Intel PIII CPU's with 1GB of memory. The nodes are connected by Myrinet as well as Gigabit Ethernet. As processor affinity cannot be specified in this version of Linux, we have used only up to 61 nodes, which leaves one node for the master and two spare nodes. The cluster is non-dedicated and as such it is not the case that all nodes run at the same speed due to interference from other jobs.

Several binary integer linear programming problems taken from the MIPLIB [2] and MP-TESTDATA [7] libraries are used in our experiments. We experiment with all solutions depth first search and also best solution search. All solutions is used to avoid any effects such as superlinear speedup which can occur with being lucky with single solution search because parallel search with different numbers of workers can investigating the search tree in a different order. For best solution search, two variant of branch and bound (B&B) are used: (i) the normal case where any new improving bound found by a worker is propagated through the worker tree; and (ii) "*noshare*" where workers do not prop-

agate bounds to other workers and only use bounds found within their own search. Noshare B&B trades off reducing communication cost with reduced pruning. Since branch and bound is expected to prune more of the tree for the same problem, the smaller problem sizes are used with backtrack search while the larger problems with branch and bound. A cut-off depth of 15 is used in backtrack search, and 20 in B&B search.

4.1. Experimental Speedup Results

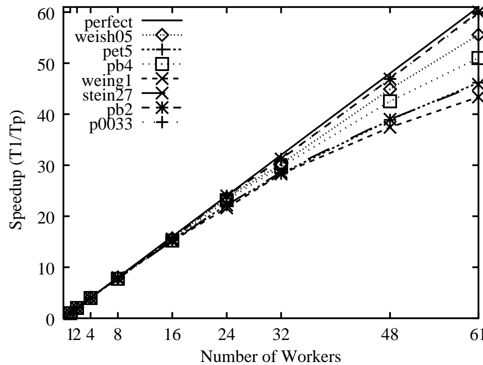

Figure 3. Speedup of backtrack search

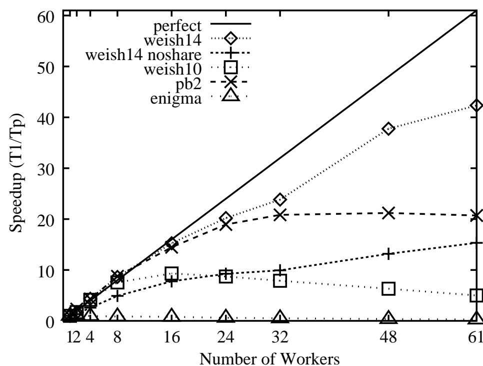

Figure 4. Speedup of B&B search

The speedup graphs for backtrack and B&B search are shown in Figures 3 and 4 respectively. The speedup with backtrack search is consistent with our analysis — near linear speedup is obtained for large enough problems. The relationship between problem size and speedup is illustrated in Fig. 3. A large problem (p0033) gives close to linear speedup even at 61 nodes while small problems with less variables (stein27 & weing1) depart from linearity at 24 nodes.[1]

In B&B search, the pruning generally decreases the search space dramatically (e.g. the search tree size is reduced by more than 100 times in B&B for pb2). The relatively small search space means that there is insufficient parallelism in the problem. This is clearly reflected in Fig. 4, which shows smaller speedups than backtrack search in Fig. 3, and the speedup lines curve down much earlier.

Some speedup anomalies are observed in Fig. 4. Enigma does not exhibit any speedup because the bounds and constraint satisfiability prunes the problem very early even on 1 worker. Adding more workers only adds to the steal overhead causing slowdown. The effect of bounds sharing to speedup is also depicted in Fig. 4 if we compare the curves of weish14 and weish14-noshare. It is clear that bounds sharing results in better speedup.

4.2. Experimental work stealing results

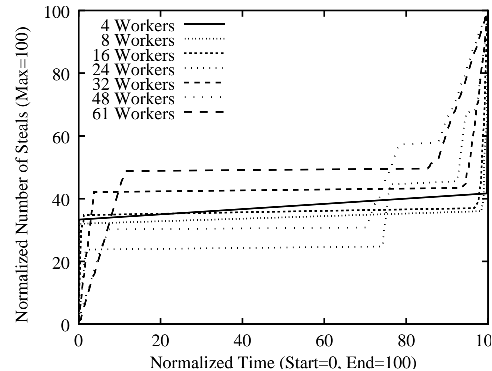

Figure 5. Work stealing over time for weish05

Figures 5 and 6 shows the percentage of steals as a function of time with both axes normalized to 100% for weish05 (backtrack) and weish14 (B&B). This is meant to illustrate the effectiveness of the work stealing strategy. The behavior of weish05 (backtrack) can be explained as follows: first there is a sharp increase in steals as workers do global steals at the start of computation, followed by a horizontal plateau with no steals and then a second increase at the end when the amount of work per worker is decreasing there are more steals again. So for a large part of the computation only a small amount of additional work stealing is needed, as indicated by the portion of the graph which is close to horizontal. This means that the overhead incurred for the dis-

1 The worst case size of the search tree may be exponential in the number of variables, so a small change in the number of variables can significant decrease inherent problem parallelism.

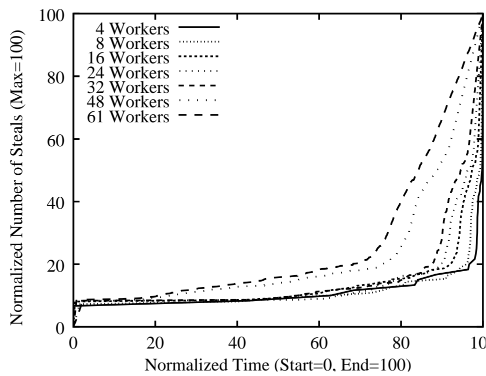

Figure 6. Work stealing over time for weish14

tributed search architecture the major part of the computation is small.

Where the number of workers is not a power of two, it can be seen that the second increase in steal rate occurs earlier as shown for the 24 and 48 worker curves which are shifted more towards the left. This is because when the number of workers is not a power of two, some workers receive less work than others and hence finish earlier which means they start stealing earlier. In B&B, Fig. 6, steals happen more frequently since the search may be terminated earlier by a broadcast bound. Thus instead of a horizontal plateau there seems to be a slow increase in the number of steals for weish14. Steals happen more frequently and earlier at the end since the B&B is more unbalanced.

5. Conclusion

We have presented a distributed framework to implement depth-first search with a general architecture based on constraint search trees. In particular, it is easy to employ either a simple solver or make use of sophisticated constraint solvers based on consistency or integer linear programming. At the heart of our framework is an adaptive architecture which can deal with a dynamically changing set of workers coming/leaving. Our main result is, under ideal conditions, such an architecture is effectively scalable and is able to achieve near linear speedup over a useful working range in the number of workers on problems with sufficient parallelism. This result is validated by our experiments which show that for a range of optimization problems that good scaling and load balancing is achieved for up to 61 workers. Thus, we demonstrate empirically that distributed search scales well to a larger number of workers than recent work [11, 13], and we provide also theoretical guarantees.

References

[1] J. E. Baldeschwieler, R. D. Blumofe, and E. A. Brewer. ATLAS: An infrastructure for global computing. In *The Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications*, 1996.

[2] R. E. Bixby, S. Ceria, C. M. McZeal, and M. W. P. Savelsbergh. MIPLIB 3.0. http://www.caam.rice.edu/~bixby/miplib/miplib.html, Jan. 1996.

[3] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual IEEE Conference on Foundations of Computer Science (FOCS'94)*, 1994.

[4] R. Finkel and U. Manber. DIB - a distributed implementation of backtracking. *ACM Transactions of Programming Languages and Systems*, 9(2):235–256, April 1987.

[5] A. Grama and V. Kumar. A survey of parallel search algorithms for discrete optimization problems. *ORSA Journal of Computing*, 7(4):365–385, 1995.

[6] G. Gupta, E. Pontelli, K. A. M. Ali, M. Carlsson, and M. V. Hermenegildo. Parallel execution of prolog programs: a survey. *ACM Transactions on Programming Languages and Systems*, 23(4):472–602, 2001.

[7] J. Heitkötter. MP-TESTDATA: SAC-94 suite of 0/1-Multiple-Knapsack problem instances. http://elib.zib.de/pub/Packages/mp-testdata/ip/sac94-suite/index.html, Nov. 1999.

[8] R. Karp and Y. Zhang. Randomized parallel algorithms for backtrack search and branch-and-bound computation. *Journal of the Association for Computing Machinery*, 40(3):765–789, July 1993.

[9] M. O. Neary, A. Phipps, S. Richman, and P. R. Cappello. Javelin 2.0: Java-based parallel computing on the internet. In *Proceedings of 6th International Euro-Par Conference*, pages 1231–1238. Springer, 2000.

[10] C. Schulte. Comparing trailing and copying for constraint programming. In *Proceedings of the International Conference on Logic Programming, ICL P*, pages 275–289. The MIT Press, 1999.

[11] C. Schulte. Parallel search made simple. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, 2000.

[12] R. V. van Nieuwpoort, T. Kielmann, and H. E. Bal. Satin: Efficient Parallel Divide-and-Conquer in Java. In *Euro-Par 2000 Parallel Processing*, number 1900 in Lecture Notes in Computer Science, pages 690–699. Springer, Aug. 2000.

[13] K. Villaverde, E. Pontelli, H.-F. Guo, and G. Gupta. PALS: An or-parallel implementation of prolog on beowulf architectures. In *Proceedings of 17th International Conference on Logic Programming, ICLP 2001*, pages 27–42. Springer, 2001.