General Distributed Backtracking Framework for Solving Combinatorial Constraint-Satisfaction Problems

David Vulakh

INTRODUCTION

Many mathematical and practical problems require elements of a set to be arranged in a way that satisfies specific conditions. Familiar examples include Sudoku, timetabling, scheduling, and resource distribution (for example, floor planning). (Stanford, 2011) Such problems, called constraint-satisfaction problems, are solved when values from a domain are assigned to variables in a way that does not violate a set of constraints. Solutions to constraint-satisfaction problems with a finite number of variables and finite domains can be found with the generalizable algorithm of recursive search with backtracking, which runs in nondeterministic polynomial time (Stanford, 2011). In order to complete large cases quickly, the computational resources of multiple cores and machines can be pooled.

Project Goal: Develop a general backtracking framework that accelerates the search for solutions to any given combinatorial constraint-satisfaction problem by distributing work to local and remote processors.

BACKGROUND

A ***Constraint-Satisfaction Problem*** is defined by the triple of sets $\langle X, D, C \rangle$

- $X = \{X_1, \ldots, X_n\}$ is the set of ***variables***
- $D = \{D_1, \ldots, D_n\}$ is the set of the respective ***domains*** of the variables in X
- $C = \{C_1, \ldots, C_m\}$ is the set of ***constraints***
  - Each constraint C_j is defined as a pair $\langle t_j, R_j \rangle$
  - $t_j \subseteq X$ is a subset of k of the variables in X
  - R_j is a Boolean relation of those k variables on the corresponding domains
- An ***evaluation*** of the variables is a function from a subset of variables to values in their respective domains. An evaluation:
  - ***satisfies*** constraint C_j iff the values assigned to the variables in t_j satisfy R_j
  - is ***consistent*** iff it satisfies all constraints in C
  - is ***complete*** iff it includes all variables in X
  - is a ***solution*** iff it is consistent and complete

(Berkley, n.d.)

A ***Costas Array*** of size M is a permutation of the integers from 1 to M the binary permutation matrix of which contains no equal displacement vectors between pairs of distinct elements (Drakakis, 2010). It is formally described by the constraint-satisfaction problem

$$\begin{aligned}&\langle\ \{X_1, \ldots, X_M\},\\ &\ \{\{1, \ldots, M\}, \ldots, \{1, \ldots, M\}\},\\ &\ \{\ \langle X, \forall i, j\ \ i \neq j \rightarrow X_i \neq X_j\rangle,\\ &\ \ \langle X, \forall i, j \nexists k, l : i \neq j \neq k \neq l \wedge k - i = l - j \wedge X_k - X_i = X_l - X_j\rangle\ \}\ \rangle\end{aligned}$$

If a distributed program with N workers solves a constraint-satisfaction problem in T time and a single worker solves it in T_0 time, the ***speedup*** S and ***efficiency*** E of the program for that problem are

$$S = \frac{T_0}{T} \qquad E = \frac{S}{N} = \frac{T_0}{NT}$$

If a distributed program with N workers solves a constraint-satisfaction problem in T time and the workers consume total ΣT time, the ***absolute overhead*** H and ***relative overhead*** H_R are

$$H = T - \frac{\Sigma T}{N} \qquad H_R = \frac{H}{T} = 1 - \frac{\Sigma T}{NT}$$

The performance of a distributed program is considered optimal if $E \geq 1$ and $H_R = 0$. In this case, the program does not consume more resources in solving a problem than do its workers or than would a single worker solving the problem independently.

DESIGN GOAL AND METHODS

The goal of the project was to create a distributed backtracking program that:

- Solves constraint-satisfaction problems
- Distributes work over multiple local and remote processor cores
- Produces files for restarting without loss of work and visualizing search statistics
- Solves the problem of Costas Arrays with $E \geq 1$

The engineering design started with a simple, local distributed commander. Over the course of several months, this commander was debugged, optimized, and expanded to produce and read restart files. After several weeks of continuous testing on large cases (through size $M = 18$), a new version of the program that distributes work over the internet was developed. A detailed description of the research process can be found in the accompanying journal.

Four programming languages were utilized to accomplish the design goal:

- ***Go*** was used for the commander program
- ***clingo***, an ASP language, was used for the worker unit
- ***Perl*** was used in a post-processor to interpret the clingo output
- ***Java*** was used to generate visualizations from the restart files produced by the Go commander

These programs have been run on four computers:

- A personal computer with 4 cores
- A university-operated machine with 8 cores
- A university-operated machine with 80 cores
- A university-operated machine with 256 cores

FRAMEWORK DESIGN

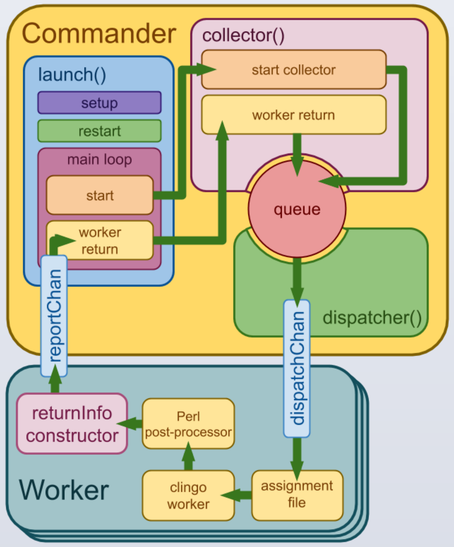

Figure 1: A schematic of the information flow through the single-machine, multi-thread implementation of the commander.

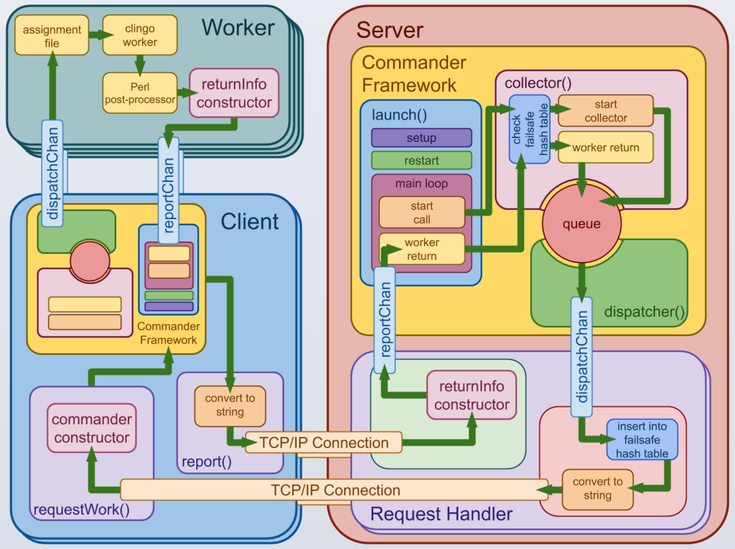

Figure 2: A schematic of the information flow through the hierarchical, multi-machine implementation of the commander.

Figure 1 displays the flow of information through the single-machine multi-thread implementation of the commander framework.

- The search begins with the launch() method of the commander object, which constructs an initial set of assignments and passes it to start collectors.
- These collectors break initial assignments into smaller subtrees and add them to the assignment queue to be searched by workers.
- A parallel dispatcher thread empties the arbitrary-length assignment queue into the static-size dispatchChan, which is read by worker threads. This intermediate dispatcher thread allows the start collectors to continue the DFS of the search tree while the dispatch channel is blocked.
- When a worker receives an assignment through the dispatch channel, it calls the clingo program and Perl post-processors to search the subtree defined by the assignment.
- Information about the results of the search (number of solutions found, exit status, runtime) is encapsulated in a returnInfo structure and pushed onto reportChan.
- The main loop empties reportChan into collector threads, which add solutions from completed searches to the total and enqueue the assignment node's children if the clingo program was killed by the --time flag.

These parallel processes are monitored from the main loop of the launch() method, which tracks end conditions and controls the collection of returnInfo.

Figure 2 depicts the flow of information through the internet commander framework, which is composed from a hierarchy of commander frameworks

- Each commander internally resolves the distribution of work as discussed to the left.
- Commanders located on client machines initiate a TCP connection to the server to request work.
- Parallel request handlers read from the dispatchChan of the server commander in the place of worker threads and send assignments to the subordinate commanders, which use the available cores of their respective client machines to solve the subproblems assigned to them.
- When a client commander finishes its subproblem, it initiates another TCP connection to the server and sends the search results to the main commander.
- The results are read by a request handler, encapsulated in a returnInfo structure, and pushed onto the reportChan.
- After the connection is severed, the client restarts and sends another request for work.

The server commander keeps a failsafe map of the worker IDs to which any given uncompleted assignment has been dispatched. The map allows the commander to redispatch assignments that are taking an unexpectedly long time to complete (perhaps because of a crash on one of the client machines) and detect and discard duplicately submitted answers.

VISUALIZATIONS

The visualizations in Figure 3 display the number of solutions found in subtrees of an $M = 16$ Costas Array search at various depths. The gradient of solutions found across all visualized subtrees is mapped to the color spectrum (from red as the minimum to blue as the maximum). A subtree defined by permutation P with last element v for $M = R \times C$ (for $M = 16$, $M = 4 \times 4$ was used) is positioned in the v^{th} cell of the rectangle of its parent permutation in the search tree, where the upper-left corner is cell 1 and numbering continues to the right and down. Invalid nodes (such as the upper left corner in all generations past 1, which represents a permutation starting with 1,1 ...) are left gray.

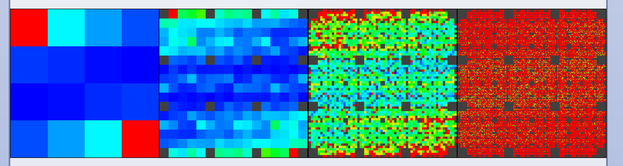

Figure 3: Visualizations of solution density at various depths of the search tree for Costas Arrays of size $M = 16$. From left to right: solution density at depth 1, 2, 3, 4.

DATA AND CONCLUSIONS

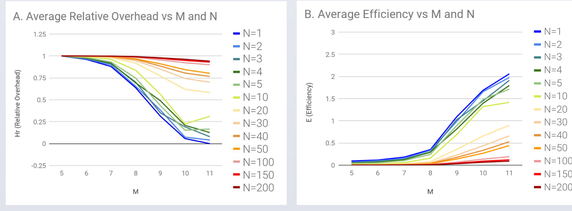

Figure 4: Graphs of commander performance. (A) A graph of average H_R with respect to M. H_R decreases as M increases, especially for smaller values of N. (B) A graph of average E with respect to M. E increases as M increases, especially for smaller values of N.

Figure 4(A) indicates that H_R decreases as M increases for $N \leq 10$.

- As $H_R = \frac{H}{T}$ and T increases with M, the nonlinear decrease of H_R with respect to M implies that H grows more slowly than T.
- The absolute overhead is dominated not by the $O(M!)$ worker calls made, but by some slower-growing (perhaps constant startup) operations.

Figure 4(B) indicates that E increases as M increases for $N \leq 10$. For sufficiently large M, E can exceed 1 for $N = 1$.

- In these cases, the commander running with one thread is faster than just the thread by itself, leading to the conclusion that the clingo workers, while capable of quickly traversing relatively shallow subtrees, are significantly slower when traversing deeper ones.
- As a result, a program splitting work for the clingo program has a 'baseline' efficiency that results from the improved speed of the worker on the shallower tree.

Both graphs indicate that the performance of the commander, evaluated both through E and H_R, improves as problem size increases.

- Constant or polynomial startup operations are apparently responsible for the majority of overhead operations.
- The trend of improving performance will likely continue.

The performance data collected indicate that the designed commander satisfies the goal of solving the problem of Costas Arrays with $E \geq 1$, as it does so frequently for $M \geq 9$. It also satisfies the additional goals of production of restart files and visualization of search statistics.

REFERENCES

Berkley. (n.d.). *Constraint Satisfaction Problems.* Retrieved from http://aima.cs.berkeley.edu/newchap05.pdf

Drakakis, K. (2010). *An Introduction to Costas Arrays.* Retrieved from http://www1.spms.ntu.edu.sg/~ccrg/documents/basicTalkSingapore.pdf

Stanford. (2011). *Constraint Satisfaction Problems.* Retrieved from https://web.stanford.edu/class/cs227/Lectures/lec14.pdf

All figures produced by student.