Constraint Programming MapReduce'd

Nikolaos Pothitos
pothitos@di.uoa.gr

Panagiotis Stamatopoulos
takis@di.uoa.gr

Department of Informatics and Telecommunications
National and Kapodistrian University of Athens
Panepistimiopolis, 157 84 Athens, Greece

ABSTRACT

While Constraint Programming (CP) aims to explore efficiently large search trees, MapReduce (MR) is a framework that focuses on huge databases and text files. In this work, we try to bridge these two cutting-edge paradigms in order to solve Constraint Satisfaction Problems (CSPs) in distributed and/or parallel environments. We implement a CP system that is responsible to distribute the parts of the serial search tree to the workers in order to explore it. MR is responsible to allocate the physical resources of a cluster and assigns them to the workers. The CP and MR phases interchange and this novel coordination results into significant speedups while solving CSPs. Empirically, our approach is applied on common CSPs, such as N Queens and Number Partitioning.

CCS Concepts

•Theory of computation → Constraint and logic programming; MapReduce algorithms; •Networks → *Cloud computing;*

Keywords

parallelization; distribution; search tree; cloud

1. INTRODUCTION

MapReduce and Constraint Programming can be linked via the word *"huge"*. MapReduce targets *huge* databases, while Constraint Programming traverses a *huge* number of candidate/partial solutions of a problem.

1.1 The MapReduce Framework

In the early Computer Science era, parallelization was mainly a research topic, because parallel and distributed infrastructures were available only in data centers and educational institutes. About a decade ago, multi-core processors were incorporated inside almost every computer and the parallelization of common sequential processes became crucial. Nowadays, it is common to utilize the resources of a cloud or even to rent whole cloud infrastructures at an affordable price [8]. In this direction, *MapReduce* (MR) is a capable framework of orchestrating thousands of commodity PCs to manage huge databases. It was invented by Google to process the whole internet archives [3].

More prominent MapReduce applications have to do with the production of large social networks graphs [1, 5] and the automatic detection of epidemics by processing large statistical data [6].

SETN '16, May 18-20, 2016, Thessaloniki, Greece
© 2016 ACM. ISBN 978-1-4503-3734-2/16/05...$15.00
DOI: http://dx.doi.org/10.1145/2903220.2903248

1.2 Parallelizing CP in Related Work

There is no doubt that there have been developed many methods to parallelize search in CP. A successful methodology is *work stealing,* in which each worker that has run out of work (nodes to be searched) asks the other workers to borrow some work [2]. Another approach is `SelfSplit`, in which each worker decides on its own which part of the search tree it will explore [4]. A more close approach to ours is decomposing a problem into many subproblems and then assign each subproblem to a worker. More formally, with this methodology each CSP is approached as an *embarrassingly parallel problem* [11, 12].

In our approach, we decompose the search tree into many parts, we create a file containing these splits and then a MapReduce system is responsible to distribute these search tree parts to the workers. There are many benefits of employing MapReduce.

- A MapReduce system will allocate by itself the CPU cores and/or the machines in a cluster of computers.
- From the above, it is obvious that MapReduce supports both parallel and distributed environments.
- It is more easy and secure for a cloud administrator to provide access to their machines through a MapReduce interface, instead of allowing the implementation of ad hoc communication strategies among the machines.
- The no-communication between mappers restriction makes MapReduce highly scalable and capable of utilizing huge data centers.

These are some of the reasons why we leverage on a plain MapReduce approach.

2. SPLITTING THE SEARCH TREE

The main objective in this work is to develop a method to partition an arbitrary search tree into (almost) equal pieces.

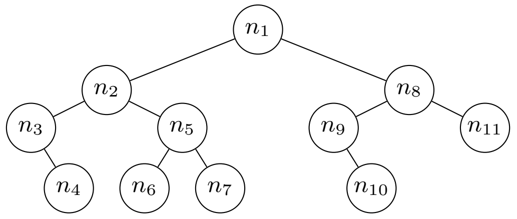

Figure 1: Labels correspond to the nodes visit order

Table 1: The *Time* when each visit to a *Node* was completed by a serial method

Node	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9	n_{10}	n_{11}
Time	2	3	4	7	9	11	12	15	20	21	22
Duration	2	1	1	3	2	2	1	3	5	1	1

Each piece can be then assigned to an independent worker and the whole search tree will be explored in parallel by the workers.

2.1 Traversing a Whole Search Tree

A sequential search method traverses one by one the nodes of the search tree, e.g. in the order they are labeled in Fig. 1. Note that this figure displays a small search tree that is *incomplete,* in the sense that an intermediate node (like n_3) may have less than two children-nodes (n_3 has only n_4 as a child).

If the total time needed to traverse the search tree in Fig. 1 is 22 time periods, Table 1 contains the indicative time when each node was visited by a serial search method.

If Table 1 was known a priori, it would have been easy to split the traversal from n_1 to n_{11}, into three streams, e.g. n_1–n_4, n_5–n_8, and n_9–n_{11}, which have three almost equivalent total durations $2+1+1+3=7$, $2+2+1+3=8$, and $5+1+1=7$.

Each range n_{begin}–n_{end} can be traversed independently by a worker in parallel. Hence, if we have three workers, we can traverse the search tree in almost 22/3 time periods, which is the total serial time divided by the number of the workers.

The symbol n_{begin} actually describes a path. I.e., it is a string with numbers like "2 1 5 3", which means "begin with the root, go to its second child (branch), then follow its first child, then follow the fifth child and reach the third child".

The above almost optimal way of splitting the search tree into equal parts and then distribute the n_{begin}–n_{end} ranges to independent workers has obviously a serious drawback: We must initially traverse the *whole* search tree to get the ranges; but what we initially wanted was to parallelize this process.

2.2 Search Tree Nodes Random Sampling

Instead of exploring the whole search tree and then split it, we can traverse a proportion of it. Our partial traversal does not need to know anything a priori about the search tree. It does not need to know its size, height, etc. It will simply traverse a part of it.

This can be accomplished by overriding ("deleting") a proportion of the search tree nodes and by constructing a table like Table 1, that will contain fewer nodes than the original one. Sampling is not a straightforward process as it raises two critical issues:

1. If a node n_i is overridden (passed by), how its time slice is replaced?

   For example, if we override n_6 in Table 1, what would be the visit time for n_7? Overriding n_6 does not mean to completely ignore n_6; its corresponding duration should be added to the visit time of n_7, because the new reduced table visit times should be as close to Table 1 times as possible.

2. Deciding to override a node n_i leads us inevitably to override all of its offspring too. E.g., the decision to override let's say n_9 in Fig. 1 is in fact a tough one: By overriding n_9 we override its offspring/descendant n_{10} too.

But let's start sampling without considering the above issues initially.

Case 1. Let R_i be a randomly generated real number, uniformly distributed in the range $[0,1]$. Let n_i be a tree node without descendants and p the *simulation factor,* i.e. the minimum proportion of the nodes we want to override. Then, n_i is overridden if $R_i \leq p$.

This means that a node with no descendants is overridden with probability at least p. We say "at least", because the precise probability to override n_i must additionally include the probability that one of n_i's ancestors is overridden.

Now we should consider what happens if the node n_i has, let's say, d descendants. Note that the term "descendants" refers not only to the nodes (children) directly connected to n_i, but also to all the nodes that belong to the sub-tree rooted in n_i (the children of the children etc.). In this case if we override n_i with probability p, this will override its descendants too. However, what we initially wanted was to override *each separate node* with probability p. Therefore, the probability to override n_i and its d descendants should be:

$$\Pr[_{n_i \text{ overrid.}}] \cdot \Pr[_{1^{\text{st}} \text{ desc. overrid.}}] \cdots \Pr[_{d^{\text{th}} \text{ desc. overrid.}}] \quad (1)$$

This is at least $p \cdot p^d = p^{1+d}$.

Case 2. Let R_i be a randomly generated real number, uniformly distributed in the range $[0,1]$. Let n_i be a tree node with d descendants and p the minimum proportion of the nodes we want to override. Then, n_i is overridden if $R_i \leq p^{1+d}$.

This is a simple workaround for the 2nd issue above that also guarantees that the average proportion of the overridden nodes will be at least p.

2.3 Pre-estimating the Descendants of a Node

Case 2 does not completely resolve the 2nd issue. When we are about to decide if a node will be overridden or not, with probability p^{1+d}, we should know the descendants number d. But this is not possible a priori, because we have not yet traversed the very node itself!

The solution is to make a *pre-estimation* of d, based on the previous history. In order to step forward in CP-MR, we make the following general assumption.

> **Assumption.** Each node is expected to have a similar descendants number and a similar time duration to the other nodes that belong to the

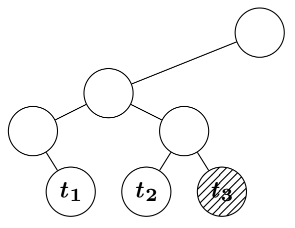

Figure 2: The third leaf-node will be simulated

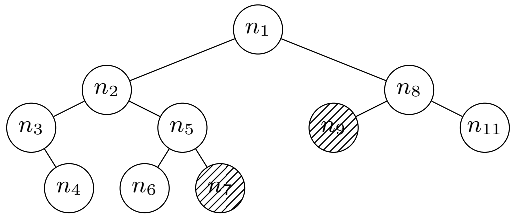

Figure 3: The tree simulation "pruned" some nodes

same tree level. In other words, the nodes that have equal distance from the root are expected to have similar number of descendants and duration.

Take for example the lowest leaf nodes in Fig. 2. The node with the label t_3 is examined on whether is going to be simulated. At first, we need to pre-estimate its descendants number d. According to the above Assumption, t_3 node will have a similar d with the other nodes in the same level (t_1 and t_2). Each of these has zero descendants. Consequently, the average $\overline{d}$ is also $(0+0)/2 = 0$.

The Assumption is not always valid and the nodes of the same level may have fluctuations regarding their descendants number. To make a better estimation for d we also take into account the respective standard deviation σ:

$$d = \overline{d} + \sigma. \tag{2}$$

With the above equation we expect that d will be less than at about 84% of the existing descendants values, according to the 68–95–99.7 statistics rule.

Hence, t_3 node will be overridden with probability $p^{1+0} = p$. And here comes the 1st issue: If the node is indeed overridden, how its simulated duration t_3 will be computed?

The duration is computed exactly in the way that we computed d: as the average of the existing non-simulated nodes in the same level: $t_3 = \sum_{i=1}^{2} t_i/2$.

Case 3. If we want to override a node n_j, its *virtual time duration* is estimated as $t_j = \frac{\sum_{i=1}^{j-1} t_i}{j-1}$. The sum refers to the nodes n_i in the same level with n_j that have not been simulated themselves. The *descendants* number d_j of n_j can be pre-estimated exactly in the same way: $d_j = \frac{\sum_{i=1}^{j-1} d_i}{j-1}$.

Returning to our example, the overall goal was to "prune" some nodes while traversing the search tree, as in Fig. 3. Again, in this case, pruning does not mean ignoring a node, but estimating its real duration. The estimation is then used to construct a virtual table like Table 2 that will be used to split the search tree nodes into almost equal parts, without having to traverse all the nodes at first.

Table 2: The virtual time when each node is visited if we override n_7 and n_9

Node	n_1	n_2	n_3	n_4	n_5	n_6	n_8	n_{11}
Time visited	2	3	4	7	9	11	$14 + t_3$	$15 + t_3 + T_3$
Duration	2	1	1	3	2	2	3	1

2.4 Sampling and Exploration Interchange

In the attempt to solve a CSP, the sampling (simulation) and exploration (solution) phases can be interchanged. Firstly, a primary simulation of the search space is made. The splits of the search space are passed to the mappers-solvers. When the solution phase of some solvers takes long time, they stop it and restart the simulation phase for the search space that remains to be explored. All the splits from the mappers are then collected and another MR round begins. The input for the new MR round is the new splits. Each time there is a delay (timeout) to some mappers, new splits are produced and a new MR round is launched. A timeout can happen e.g. when the *real* time needed to search a tree part is a multiple of ten of the *expected* time.

3. EXPERIMENTAL EVALUATION

Our CP-MR framework was employed to solve the N Queens problem. The goal in this problem is to place N queens on a $N \times N$ chessboard so that they do not attack each other. For each N Queens instance we found *all* the solutions.

In the same context, we solved the N Number Partitioning problem too. In this CSP, given a set with the first N numbers, we try to collect all the disjoint subset pairs S_1 and S_2, that satisfy the conditions $|S_1| = |S_2|$, $\sum_{i \in S_1} i = \sum_{i \in S_2} i$, and $\sum_{i \in S_1} i^2 = \sum_{i \in S_2} i^2$.

The sequential search methods for these problems can be found in Naxos Solver, a CP library written in C++ [10]. The sequential approaches were modified to read the input which is lines in the form: n_{begin}–n_{end}. The Solver starts exploring the search tree from the node n_{begin} until it reaches n_{end}. The source code is freely available at http://di.uoa.gr/~pothitos/CPMR

The Naxos Solver's executable that reads n_{begin}–n_{end} from its input was provided to Hadoop Streaming 2.7.1, which is a MapReduce system. Each Naxos Solver's executable plays the role of a Hadoop mapper-worker. The reducers do not play actually any role: They simply echo the solutions they get.

Hadoop was installed on a cloud infrastructure [9] that consists of 8 Ubuntu Linux 14.04 virtual machines with 8-core processors at 2 GHz and 8 GB memory. The exact CPU specifications are unknown due to the so-called cloud virtualization. The first machine (the Hadoop master) has a 60 GB hard disk and the other seven machines (the Hadoop slaves) have a 40 GB disk each.

In order to produce the search tree partitions for each N, we used a simulation factor equal to 99.9%. The created file with the partitions was the input for the CP-MR system.

Figure 4 illustrates the speedups gained while producing all the N Queens solutions versus the mappers-workers employed by the CP-MR system. In our time measurements we included all the CP-MR steps and, of course, the simulation process.

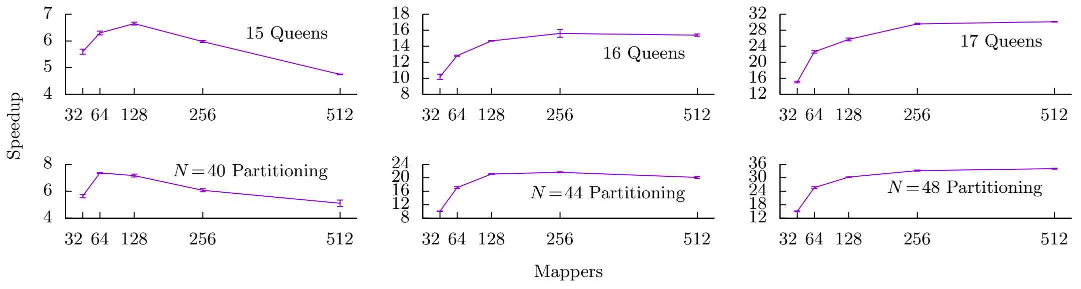

Figure 4: The speedup gained vs. the number of the mappers employed

We repeated our experiments three times and recorded the *standard error* (SE) between the runs initiated with different random seeds. The standard error (SE) is defined as the ratio of the standard deviation (SD) of the recorded times to the square root of the number of samples, i.e. $\mathrm{SE} = \mathrm{SD}/\sqrt{n}$, where $n = 3$ repetitions. The standard error is depicted as an "I" on top of each measurement in the following figures, just to get an idea of the possible variance between the samples. The standard error is not visible in most figures because it is small.

As the mappers number increases until 128, the improvements in time and speedup are more dramatic than the improvements gained until we reach 512. An important reason for this is that we have only 64 cores available every moment. Even if we employ 512 mappers, only 64 of them will run simultaneously.

It is also worth to note that for the smaller N values the speedups are lower. This has to do with the significant Hadoop overhead when the sequential time to solve the CSP is relatively small. On the other hand, the speedups for the larger CSP instances almost reach 35.

4. CONCLUSIONS AND FUTURE WORK

This work is an attempt to bridge two different but commonplace paradigms: CP and MR. To accomplish this connection, we partitioned the search tree, we recorded the partitions into a text file, and we forwarded these partitions to a MR system. The MR system invoked many mappers that played the role of a CP solver. Each CP solver was assigned by the MR system different search tree partitions to explore.

The future directions for this hybrid approach are many. Most notably, we can employ multiple MR rounds to solve optimization problems. In these problems, the goal is not to find all the solutions, but to get the *best* solution according to specific criteria. In this case, the reducers' role will be more active, as they should gather all the solutions and output the best one.

5. ACKNOWLEDGMENTS

This research was partially funded by the University of Athens Special Account of Research Grants no 10812.

In this work we used the ~okeanos [7] cloud infrastructure provided by the Greek Research and Education Network http://okeanos.grnet.gr

6. REFERENCES

[1] F. Afrati, D. Fotakis, and J. D. Ullman. Enumerating subgraph instances using Map-Reduce. In *ICDE 2013*, pages 62–73, Los Alamitos, CA, USA, 2013. IEEE Computer Society.

[2] G. Chu, C. Schulte, and P. J. Stuckey. Confidence-based work stealing in parallel constraint programming. In I. P. Gent, editor, *CP 2009*, volume 5732 of *LNCS*, pages 226–241. Springer, Berlin Heidelberg, 2009.

[3] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI 2004*, pages 137–149, 2004.

[4] M. Fischetti, M. Monaci, and D. Salvagnin. Self-splitting of workload in parallel computation. In H. Simonis, editor, *CPAIOR 2014*, volume 8451 of *LNCS*, pages 394–404. Springer International Publishing, Switzerland, 2014.

[5] M. Gergatsoulis, C. Nomikos, E. Kalogeros, and M. Damigos. An algorithm for querying linked data using Map-Reduce. In *6th International Conference, Globe 2013: Data Management in Cloud, Grid and P2P Systems, Prague*, volume 8059 of *LNCS*, pages 51–62. Springer, 2013.

[6] J. Ginsberg, M. H. Mohebbi, R. S. Patel, L. Brammer, M. S. Smolinski, and L. Brilliant. Detecting influenza epidemics using search engine query data. *Nature*, 457(7232):1012–1014, 2009.

[7] E. Koukis and P. Louridas. ~okeanos IaaS. In *Proceedings of Science EGICF12-EMITC2: EGI Community Forum 2012/EMI Second Technical Conference, Munich, Germany*, 2012.

[8] V. Koukis, C. Venetsanopoulos, and N. Koziris. ∼okeanos: Building a cloud, cluster by cluster. *IEEE Internet Computing*, 17(3):67–71, 2013.

[9] V. Koukis, C. Venetsanopoulos, and N. Koziris. Synnefo: A complete cloud stack over Ganeti. *USENIX ;login:*, 38(5):6–10, October 2013.

[10] N. Pothitos. NAXOS SOLVER. http://di.uoa.gr/~pothitos/naxos, 2015.

[11] J.-C. Régin, M. Rezgui, and A. Malapert. Embarrassingly parallel search. In C. Schulte, editor, *CP 2013*, volume 8124 of *LNCS*, pages 596–610. Springer, Berlin Heidelberg, 2013.

[12] J.-C. Régin, M. Rezgui, and A. Malapert. Improvement of the embarrassingly parallel search for data centers. In B. O'Sullivan, editor, *CP 2014*, volume 8656 of *LNCS*, pages 622–635. Springer International Publishing, Switzerland, 2014.