

# Using Answer Set Programming for Assigning Tasks to Computing Nodes

Franz Wotawa<sup>1,\*</sup>

<sup>1</sup>Graz University of Technology (TU Graz), Institute of Software Technology, Inffeldgasse 16b/2, A-8010 Graz, Austria

## Abstract

Allocating tasks to computing nodes in a network is an important configuration problem. In the case of fail-safe networks, such configuration must be changed during operation if a computing node fails. Hence, a fast configuration is required. In this paper, we formulate a tasks-to-computing-nodes assignment problem and its constraints using answer set programming. We performed an initial experimental evaluation utilizing several smaller to mid-size problem instances to show whether logic reasoning based on answer set programming is feasible for practical applications. We discovered that reasoning is fast if a solution exists but not when there is no solution. Further constraints help to decide that a problem instance is unsolvable early in the search, which improves the outcome.

## Keywords

Configuring computing nodes, ASP models for configuration, Experimental analysis

## 1. Introduction

There has been plenty of work in the area of configuration and recommender systems, including service configuration [1], governance systems [2], or product configuration [3] to refer to recent work. However, answer set programming for representing models used for configuration and reasoning to obtain valid configuration has recently gained more attention, e.g., see [3, 4, 5]. In this paper, we contribute to this research direction and introduce a model and an evaluation for configuring networks comprising computing nodes for executing pre-defined tasks. The underlying problem is related to scheduling and shift designs [6].

The main motivation behind our work comes from practical applications, where, for example, tasks, i.e., programs, have to be deployed on computing nodes in a network. Although this problem can be seen as a static one that must only be solved before the operation, we may require to re-configure such a task assignment during operation whenever a computing node fails (see, e.g., [7]). Such re-configuration tasks have to be carried out under time restrictions. To evaluate whether modern reasoning methods like answer set programming can be used for this task, we conduct an experimental evaluation. This evaluation comprises several instances of the corresponding task to computing node assignment problems considering various sizes of nodes and tasks. The evaluation utilizes the answer set programming solver

clingo [8]. The question we want to answer is regarding the approach's limitations regarding the problem size. Can we use answer set programming (and in particular clingo) to provide task assignments fast enough to be used during operation?

We structure this paper as follows. First, we introduce the underlying configuration problem and clingo implementation. Afterward, we discuss the experimental evaluation, i.e., the basic setup and the results obtained. Finally, we conclude the paper.

## 2. Problem description

We start defining the task to node assignment problem. We assume we have  $k$  computing nodes  $n_1, \dots, n_k$  and  $n$  tasks  $t_1, \dots, t_n$  to be assigned to the nodes. For each node  $n_i$ , we know the maximum number of tasks  $c(n_i)$  it can hold and the available memory  $m(n_i)$ . For each task  $t_j$ , we know its memory consumption  $m(t_j)$ . From this knowledge, we obtain several constraints an assignment must fulfill to be valid. In the following, we formalize these constraints assuming that the function  $assigned(n_i)$  returns a set of tasks that is assigned to a node  $n_i$ .

First, the required memory from the tasks assigned to a node shall never exceed the available memory of this node. These constraints can be formalized as follows:

$$\forall i \in \{1, \dots, k\} : \left( \sum_{t_j \in assigned(n_i)} m(t_j) \leq m(n_i) \right) \quad (1)$$

Second, the number of tasks assigned to a node shall never exceed the maximum number of tasks the node can hold, i.e., formally, we write:

$$\forall i \in \{1, \dots, k\} : (|assigned(n_i)| \leq c(n_i)) \quad (2)$$

ConfWS'24: 26th International Workshop on Configuration, Sep 2–3, 2024, Girona, Spain

\*Corresponding author.

✉ wotawa@ist.tugraz.at (F. Wotawa)

🌐 <https://www.tugraz.at/> (F. Wotawa)

🆔 0000-0002-0462-2283 (F. Wotawa)

© 2024 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

A solution to the tasks to computing nodes assignment problem is an assignment of all tasks to all nodes such that  $\forall j \in \{1, \dots, k\} : \exists i \in \{1, \dots, n\} : t_j \in \text{assigned}(n_i)$ , there is no tasks assigned to two different nodes, i.e.,  $\forall i, j \in \{1, \dots, k\}, i \neq j : \text{assigned}(n_i) \cap \text{assigned}(n_j) = \emptyset$ , and all constraints are fulfilled. Such an assignment is a valid one and may not exist. For example, if the number of tasks exceeds the number of free slots of the nodes or if the required memory for the tasks is not provided, there is no solution. Hence, we have two necessary conditions that must hold for obtaining a solution, i.e.:

$$n \leq \sum_{i=1}^k c(n_i) \quad (3)$$

$$\sum_{j=1}^n m(t_j) \leq \sum_{i=1}^k m(n_i) \quad (4)$$

It is worth noting that similar problems have additional constraints, e.g., stating that tasks need to be together in the same computing node. We may also introduce optimality criteria like preferring solutions requiring the least amount of computing nodes. Furthermore, we may also consider variants of the problem, i.e., reconfiguration of assignments. In the context of this paper, we do not tackle such extensions. We solely focus on answering the question regarding the applicability of the answer set program to solve the problem within a reasonable amount of time.

After outlining the problem in general, we present a solution using answer set programming where we rely on the syntax of the `clingo` solver<sup>1</sup> [8], which is similar to the Prolog language. Due to space restrictions, we do not give an introduction to answer set programming (ASP). Instead, we refer to introductory literature into ASP, e.g., [9].

A `clingo` model for the node assignment problem comprises three parts. First, we define the number of computing nodes and tasks and their capacities and requirements. For every node, e.g., `n2`, we use three facts, where the predicate `tcapacity` denotes the maximum number of tasks, and `mcapacity` the maximum available memory for the given task:

```
node(n2). tcapacity(n2,1). mcapacity(n2,20).
```

For each task, e.g., `t1`, we add two facts, where the predicate `memory` is for defining the required memory of the given task to a pre-defined value:

```
task(t1). memory(t1,30).
```

Second, we generate all potential solutions. For this purpose, we introduce a predicate for a node assignment

of a task. Let us call this predicate `select` that takes a task `T` as the first parameter and node `N` as the second. In `clingo`, the generate part for the node assignment problem is given as follows:

```
{ select(T,N) : node(N) } = 1 :- task(T).
```

This rule generates a grounded predicate that assigns tasks to each computing node. Obviously, not all assignments are correct when considering the constraints. Hence, in the last part, we formalize the first two constraints of the general problem, i.e., Equations 1 and 2, but not the other Equations 3 and 4. For this purpose, we introduce a predicate `nrTasksAssigned` that holds the number of tasks that are assigned to a particular node and a predicate `memRequired` that holds the required memory for a node considering the assigned tasks. The information regarding the predicates can be obtained from the selected task for a node (`select` predicate) and the memory required for a task. For the latter, we introduce the predicate `memory`. The predicate `nrTasksAssigned` can be formalized in `clingo` as follows:

```
nrTasksAssigned(N,M) :-
  M = #count {T:select(T,N)},
  node(N).
```

Similar, we can define the `memRequired` predicate:

```
memRequired(N,M) :-
  M = #sum {NM:select(T,N), memory(T,NM)},
  node(N).
```

Using these predicates, we can formulate the constraints:

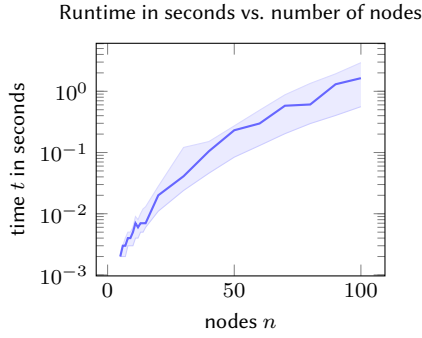
```
:- nrTasksAssigned(N,M), tcapacity(N,C), M>C.
:- memRequired(N,M), mcapacity(N,C), M > C.
```

The first constraint states that it is impossible to assign more tasks to a node than the node can hold. The second constraint states that the memory requirements of all tasks should not exceed the memory capacity of the computing node.

### 3. Experimental evaluation

The following experimental evaluation aims to investigate the runtime behavior for finding one solution of the task to computation node assignment problem using the ASP solver `clingo`. In particular, we are interested in the number of nodes that can be handled requiring less than a fixed boundary of time, e.g., 0.01 or 0.1 seconds. In the following, we outline the experimental setup and present and discuss the obtained results.

<sup>1</sup>See <https://potassco.org/about/>



**Figure 1:** Solving runtime of consistent instances

**Experimental setup:** To develop several instances of the task assignment problem, we wrote a Java program for generating such instances automatically. We ranged the number of nodes from 5 to 100. The number of tasks for each instance was randomly chosen between the number of nodes and double the number of nodes. The capacity of each node was randomly set from  $1, 2, \dots, 10$ . The memory provided by each node was randomly chosen from  $20, 40, 60, \dots, 200$ . The memory required by every task was randomly set from  $10, 20$ , and  $30$ . With this setup, we generated only satisfiable instances, i.e., problem instances where a solution exists. For a category of instances comprising  $n$  nodes, we called the instance generator 10 times. Finally, we received 200 different problem instances.

We conducted the experiments using an Apple MacBook Pro, with an Apple M1 CPU comprising 8 cores and 16 GB of main memory, running under macOS Sonoma Version 14.4.1. For computing solutions, we relied on `clingo` version 5.7.1 and applied the standard setup.

**Experimental results:** After generating the problem instances, we ran `clingo` to compute one solution, i.e., we ran `clingo` using the prompt `clingo -time-limit=10 -outf=2` for each file. Hence, we set a time limit of 10 seconds and obtained all results in JSON format. After analyzing the results for correctness, we summarized the outcome, i.e., the runtime for each category of a particular number of nodes. Figure 1 depicts the minimum, maximum, and average runtime for all 10 runs for each category.

We see that when using ASP solving utilizing `clingo` we can provide one solution even for larger instances of 100 computing nodes in a reasonable amount of time. However, when using the approach during operation, and especially for systems with harder requirements regarding answering time, e.g., real-time systems, a runtime of almost 3 seconds might not be feasible. We would like answers in less than 0.1 or 0.01 seconds for such systems,

which can be achieved for 20 or 13 nodes, respectively.

Motivated by the results, we performed further experiments, considering problems that likely cannot be solved. Unfortunately, in this case, we often ran into reaching the solving time limit of 10 seconds, even starting with small examples only considering 5 computing nodes. Instances with more than 7 nodes that might be unsatisfiable always reach the 10-second limit. For those instances where unsatisfiability could be established, the runtime varies between 0.003 and 6.255 seconds. The latter was obtained for a problem instance comprising 7 computing nodes. Hence, unsatisfiable instances can hardly be identified when considering more computing nodes, which might also be an issue for practical applications.

We carried out another experiment to tackle the mentioned problem of potential unsatisfiability. We selected 3 problem instances for which we could not compute a result. Two instances had 7 nodes, and one had 15 nodes. For these problem instances, we added further constraints that cover Equations 3 and 4. For all three `clingo` files that correspond to the problem instances, we added the following code:

```
totalCapacity(C) :- C=#sum{T:tcapacity(N,T)}.
totalMemReq(C) :- C=#sum{M:memory(T,M)}.
totalMem(C) :- C=#sum{N:mcapacity(CN,N)}.
:- totalCapacity(C), C < 21.
:- totalMemReq(Ctask), totalMem(Cnode),
   Ctask > Cnode.
```

Note that the 21 in the above code represents the number of tasks<sup>2</sup>. We adapted this value for each instance and set it to 21, 28, and 60 respectively. When running `clingo` on the three files, we obtained an immediate response of unsatisfiability. In all cases, this response was less than 0.025 seconds. Hence, adding further constraints that allow distinguishing satisfiable from unsatisfiable cases as early as possible solves the problem.

In summary, `clingo` allows for fast computation of solutions if they exist. The reasoning for the mentioned tasks-to-computing-nodes assignment problem is fast enough for at least smaller examples to ensure a timely response. However, whether this is good enough depends on the application domain. The challenges we obtained in the case of unsatisfiability can be solved by setting a time limit for `clingo` and introducing additional constraints. Results from this case study may also apply to other configuration problems.

**Threats to validity:** There are many threats to validity worth mentioning. The experimental evaluation is

<sup>2</sup>The number of tasks for a particular problem instance can also be obtained using `clingo`. The command `#count{T:task(T)}` delivers this number. However, we set the number manually for the three experiments.

limited in the number of problem instances. There might be satisfiable instances that may take longer than reported for a given number of computing nodes. However, we do not expect a very large deviation from the results. Furthermore, we only considered one rather simple configuration problem. In case of more complex problems, we expect different runtimes. Nevertheless, the effect of adding constraints to determine unsatisfiability as early as possible should still be visible. This might also hold for the observation that unsatisfiability might be hard to identify and, therefore, require more computing time. We carried out all experiments using `clingo`'s standard-setting. There might be differences to observe when changing parameters and setup. There might also be differences when considering other versions of `clingo`, the hardware, or the operating system. Finally, the representation of the problem in `clingo` might also influence the performance.

## 4. Conclusions

In this paper, we used the configuration problem of assigning tasks to computing nodes to answer whether answer-set programming is feasible for practical applications. For smaller problem instances, answer set programming might be feasible, providing a fast response within a fraction of a second. For larger instances, we may not be able to provide a solution within a reasonable answer time. Furthermore, we identified a challenge, i.e., the extended runtime required for providing an answer in case of unsatisfiability, and a solution, i.e., the effect of additional constraints on reducing the runtime. In future research, we want to extend the configuration problem to capture the case of task assignments for computing networks at runtime. For this purpose, we want to formulate a corresponding re-configuration problem. Furthermore, we want to extend the experimental evaluation using more example instances and additional constraints and consider computing optimal solutions concerning a given optimization criteria, e.g., using the least number of computing nodes.

## References

- [1] E. M. Strøm, T. M. Münsberg, L. Hvam, Identifying potential applications of service configuration systems in a logistics company, in: J. M. Horcas, J. A. Galindo, R. Comploi-Taupe, L. Fuentes (Eds.), Proc. 25th Intl. Workshop on Configuration, Málaga, Spain, September 6-7, 2023, CEUR-WS.org, 2023, pp. 60–66. URL: <https://ceur-ws.org/Vol-3509/paper9.pdf>.
- [2] S. Muñoz-Hermoso, D. Benavides, F. J. D. Mayo, Multi-level configuration in smart governance systems, in: J. M. Horcas, J. A. Galindo, R. Comploi-Taupe, L. Fuentes (Eds.), Proc. 25th Intl. Workshop on Configuration, Málaga, Spain, September 6-7, 2023, CEUR-WS.org, 2023, pp. 67–74. URL: <https://ceur-ws.org/Vol-3509/paper10.pdf>.
- [3] R. Comploi-Taupe, G. Friedrich, T. Niestroj, Dynamic aggregates in expressive ASP heuristics for configuration problems, in: J. M. Horcas, J. A. Galindo, R. Comploi-Taupe, L. Fuentes (Eds.), Proc. 25th Intl. Workshop on Configuration, Málaga, Spain, September 6-7, 2023, CEUR-WS.org, 2023, pp. 75–84. URL: <https://ceur-ws.org/Vol-3509/paper11.pdf>.
- [4] N. Rühling, T. Schaub, T. Stolzmann, Towards a formalization of configuration problems for asp-based reasoning: Preliminary report, in: J. M. Horcas, J. A. Galindo, R. Comploi-Taupe, L. Fuentes (Eds.), Proc. 25th Intl. Workshop on Configuration, Málaga, Spain, September 6-7, 2023, CEUR-WS.org, 2023, pp. 85–94. URL: <https://ceur-ws.org/Vol-3509/paper12.pdf>.
- [5] R. Comploi-Taupe, A. A. Falkner, S. Hahn, T. Schaub, G. Schenner, Interactive configuration with ASP multi-shot solving, in: J. M. Horcas, J. A. Galindo, R. Comploi-Taupe, L. Fuentes (Eds.), Proc. 25th Intl. Workshop on Configuration, Málaga, Spain, September 6-7, 2023, CEUR-WS.org, 2023, pp. 95–103. URL: <https://ceur-ws.org/Vol-3509/paper13.pdf>.
- [6] M. Abseher, M. Gebser, N. Musliu, T. Schaub, S. Woltran, Shift design with answer set programming, in: F. Calimeri, G. Ianni, M. Truszczynski (Eds.), Proc. 13th Intl. Conf. on Logic Programming and Nonmonotonic Reasoning, 2015, Lexington, KY, USA, September 27-30, 2015., volume 9345 of LNCS, Springer, 2015, pp. 32–39. URL: [https://doi.org/10.1007/978-3-319-23264-5\\_4](https://doi.org/10.1007/978-3-319-23264-5_4). doi:10.1007/978-3-319-23264-5\_4.
- [7] A. Ballesteros, M. Barranco, J. Proenza, L. Almeida, F. Pozo, P. Palmer-Rodríguez, An infrastructure for enabling dynamic fault tolerance in highly-reliable adaptive distributed embedded systems based on switched ethernet, *Sensors* 22 (2022) 7099. URL: <https://doi.org/10.3390/s22187099>. doi:10.3390/s22187099.
- [8] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Multi-shot asp solving with clingo, *Theory and Practice of Logic Programming* 19 (2019) 27–82. doi:10.1017/S1471068418000054.
- [9] T. Eiter, G. Ianni, T. Krennwallner, Answer set programming: A primer, in: S. Tessaris, E. Franconi, T. Eiter, C. Gutierrez, S. Handschuh, M.-C. Rousset, R. A. Schmidt (Eds.), Reasoning Web. Semantic Technologies for Information Systems: 5th Intl. Summer School, Brixen-Bressanone, Italy, August 30 - September 4, 2009, pp. 40–110, Springer Berlin Heidelberg, . URL: [https://doi.org/10.1007/978-3-642-03754-2\\_2](https://doi.org/10.1007/978-3-642-03754-2_2). doi:10.1007/978-3-642-03754-2\_2.