Web-scale distributed AI search across disconnected and heterogeneous infrastructures

Tom Kelsey, Martin McCaffery
School of Computer Science
University of St Andrews
KY16 9SX, United Kingdom
Email: {twk,mm689}@st-andrews.ac.uk

Lars Kotthoff
Cork Constraint Computation Centre
Department of Computer Science
University College of Cork
Cork, Ireland
Email: larsko@4c.ucc.ie

***Abstract*—We present a robust and generic framework for web-scale distributed e-Science Artificial Intelligence search. Our validation approach is to distribute constraint satisfaction problems that require perfect accuracy to 10, 12 and 15 digits. By checking solutions obtained using the framework against known results, we can ensure that no errors, duplications nor omissions are introduced. Unlike other approaches, we do not require dedicated machines, homogeneous infrastructure or the ability to communicate between nodes. We give special consideration to the robustness of the framework, minimising the loss of effort even after a total loss of infrastructure, and allowing easy verification of every step of the distribution process. The unique challenges our framework tackles are related to the combinatorial explosion of the space that contains the possible solutions, and the robustness of long-running computations. Not only is the time required to finish the computations unknown, but also the resource requirements may change during the course of the computation. We demonstrate the applicability of our framework by using it to solve challenging problems using two separate large-scale distribution paradigms. The results show that our approach scales to e-Science computations of a size that would have been impossible to tackle just a decade ago.**

***Index Terms*—Distributed computation, AI search, infrastructure, scalable computing, constraints**

I. Introduction

e-Science benefits from the provision and use of search procedures to tackle a variety of important problems. In this paper we focus on classes of e-Science problems that are discrete, computationally expensive, and require perfect accuracy to 10, 12 and 15 digits. We describe and evaluate a distribution framework that allows the deployment of combinatorial search to complex scientific problems with excellent expected performance.

In the face of the constant increase of computing power available to computing users of all levels, the processing of so-called "big data" is one of the directions in which there has been intensive research activity, and now applications can be scaled across hundreds of machines relatively easily. However, the situation in many areas of Artificial Intelligence (henceforth AI) is completely different. Distributing problems across several machines had been a research endeavour long before the advent of easily accessible computational resources and big data. The problems AI aims to solve have typically required a large amount of computational resources to solve problems of practical relevance.

Considering the keen interest of AI researchers in parallelisation, it is somewhat paradoxical that frameworks to distribute AI techniques are still in their infancy when it comes to practical applications. One such example is Apache Mahout [1], which leverages the generic Hadoop framework to distribute Machine Learning algorithms. For AI search on the other hand, there are, to the best of our knowledge, no similar frameworks. Consequentially, the system described herein is of great interest as an emergent achitecture, both to be utilised by the e-Science search community as-is, or to be expanded upon by the general community.

AI search itself has a wide range of applications and has been applied in such varied contexts as planning workflows [2], identifying optimal protein and DNA structures [3], [4], and obtaining qualitative models of dynamics systems arising in a wide range of scientific areas [5]–[11].

AI search involves the efficient creation, exploration and pruning of very large search trees (e.g. for the game of chess, the tree has an estimated 10^{47} nodes). In many cases it is acceptable to find the first solution from many candidates, or accept sub-optimal solutions with respect to a cost function to limit the amount of search performed. However, we often require either all solutions to a given problem, or a solution that has a guarantee of optimality.

Even when only the first solution is required, the time to find it can quickly grow to days, months or even years on a single computer. In most cases, this timescale is unacceptable: we must be able to find a solution more quickly. There are two strategies for achieving this: the AI search techniques can be improved to be more efficient for the problem, or the search can be distributed across several machines such that the time to find a solution decreases without actually decreasing the total effort. The framework presented in this paper pursues the latter strategy. Our requirements for such a framework can be summarised as follows.

- **Scalability.**
  We want to be able to use as many resources as possible at the same time, regardless of type and location and with minimal connectivity requirements.
- **Robustness.**
  The framework must be able to cope with hardware and similar failures. In particular, the amount of computa-

tional effort lost because of such an event should be small.

- **Verifiability.**
  In order to be useful for solving open problems, we must be able to follow each step in the distribution process to verify that AI search proceeded correctly and no solutions were lost.

In this paper we describe two frameworks that fulfil these requirements. The design and implementations are motivated by the Recovery Oriented Computing [12], [13] aspects of the much wider research into Ultralarge systems [14]. The AI search undertaken is Constraint Programming, described in Section I-A. This is not a restriction: as a generalisation of propositional satisfiablity (SAT), most AI search problems can be expressed as Constraint Programming problems. The application areas that we use to evaluate the implementation of the framework are described in Sections I-B and I-C.

A. Constraint Programming

Constraints are a natural and compact way of representing problems that are ubiquitous in everyday life. Constraint Programming investigates techniques for solving problems that involve constraints. Common application domains include other areas of Artificial Intelligence such as planning, but also real world and industrial applications such as scheduling, design and configuration or diagnosis and testing. Wallace [15] gives an early overview of application areas.

Constraint problems are typically solved by building a search tree in which the nodes are assignments of values to variables and the edges lead to assignment choices for the next variable. If at any node a constraint is violated, search backtracks by returning to a previous state. If a leaf is reached and no constraints are violated, all variables have been assigned values and this set of assignments denotes a solution to the Constraint Satisfaction Problem (CSP).

Clearly the search trees are exponential in the number of variables. Exploring all of them is infeasible in many cases and inference is used at each node of the search tree to prune values from the domains of unassigned variables that cannot be part of a solution based on the assignments made so far. Inference also allows backtracking before a constraint is violated: if the domain of a particular variable becomes empty, the set of assignments made so far cannot be part of a solution.

The inference checks have a computational cost and the trade-off is between the effort of making checks – hopefully resulting in a reduction of the search space – and the effort of searching a presumably larger tree but at a cheaper cost per node. This is an area of active research and the Handbook of Constraint Programming [16] provides more details on the many techniques that can be used to solve constraint problems.

B. Semigroups

We apply one of our frameworks to finding the semigroups of order 10, i.e. finding all ways of filling in a blank 10×10 mulitiplication table such that multiplication is associative up to symmetric equivalence. For orders less than 10, this problem can be solved by a combination of enumeration formulae and computation on a single processor. Table I – with entries taken from sequence A001423 of the On-Line Encyclopaedia of Integer Sequences – demonstrates the combinatoric growth in the number of solutions with increasing order, and motivates the use of multiple compute nodes to explore the solution space. The table for the semigroups of order n has n^2 cells, and each of these can take any one of n values. Hence the search space for order n is n^{n^2}. For the problem under consideration, $n = 10$, the size of the search space is 10^{100}. To put this number into context, it is currently estimated that there are approximately 10^{80} atoms in the universe. The search space for our problem is so vast that we cannot possibly hope to solve it by brute force search.

TABLE I
NUMBER OF SEMIGROUPS OF ORDER n. THE NUMBER FOR ORDER 10 WAS FOUND USING THE TECHNIQUES DESCRIBED IN THIS PAPER.

n	Semigroups
1	1
2	4
3	18
4	126
5	1,160
6	15,973
7	836,021
8	1,843,120,128
9	52,989,400,714,478
10	12,418,001,077,381,302,684

Method	Semigroups	CPU years
Known	1	
Formula	12,417,282,095,522,918,811	
Construction	461,919,236,210,408	
Minion	257,062,622,173,464	133
Total:	12,418,001,077,381,302,684	

TABLE II
SUMMARY OF METHODS USED FOR THE ENUMERATION OF THE SEMIGROUPS OF ORDER 10

Recent advances in the theory of finite semigroups have led to an enumerative formula [17] that gives the number of ‘almost all’ semigroups of given order. This formula has been used as a basis for studies enumerating semigroups and monoids using standard, single-processor search techniques [18]–[20]. Our constraint programming methodology was therefore to first rule out these solutions by adding additional constraints, and then implement the distributed search using the framework described in this paper. A summary of the solutions and CPU effort required is given in Table II.

A full description of the CSP model, the breaking of symmetries and the reduction into case-splits are given in [21]–[23].

C. The Progressive Party Problem

Our second class of problems was selected due to its combination of small data size, large number of solutions, and amount of CPU time needed to identify and enumerate the solutions. The Progressive Party Problem is one of a library of AI search problems [24]. We solve a modification of the problem, being all solutions to the problem of generating all instances of the Progressive Party Problem as described in [25]. Scalability, robustness and verifiability of our framework are demonstrated by finding exactly the known number of solutions – $5,782,683,648 \quad (\approx 10^{10})$ and $161,915,142,144 \quad (\approx 10^{12})$ – for selected instances.

II. Related work

The parallelisation of depth-first search has been the subject of much research in the past. The first papers on the subject study the distribution over various specific hardware architectures and investigate how to achieve good load balancing [26], [27]. Distributed solving of constraint problems specifically was first explored only a few years later [28].

Backtracking search in a distributed setting has also been investigated by several authors [29], [30]. A special variant for distributed scenarios, asynchronous backtracking, was proposed in [31]. Yokoo et al. formalise the distributed constraint satisfaction problem and present algorithms for solving it [32].

Schulte presents the architecture of a system that uses networked computers [33]. More recent papers have explored how to transparently parallelise search without having to modify existing code [34].

Most of the existing work is concerned with the problem of effectively distributing the workload such that every compute node is kept busy. The most prevalent technique used to achieve this is work stealing. The nodes communicate with each other and those which are idle request a part of the work a busy node is doing. Blumofe and Leiserson propose and discuss a work stealing scheduler for multithreaded computations in [35]. Rolf and Kuchcinski investigate different algorithms for load balancing and work stealing in the specific context of distributed constraint solving [36].

Several frameworks for distributed constraint solving have been proposed and implemented, e.g. FRODO [37], DisChoco [38] and Disolver [39]. All of these approaches have in common that the systems to solve constraint problems are modified or augmented to support distribution of parts of the problem across and communication between multiple compute nodes. The constraint model of the problem remains unchanged however; no special constructs have to be used to take advantage of distributed solving. All parallelisation is handled in the respective solver. This does not preclude the use of an entirely different model of the problem to be solved for the distributed case in order to improve efficiency, but in general these solvers are able to solve the same model both with a single executor and distributed across several executors.

The decomposition of constraint problems into subproblems which can be solved independently has been proposed in [40], albeit in a different context. In this work, we explore the use of this technique for parallelisation. A similar approach was taken in [36], but requires parallelisation support in the solver.

III. Distributing CSPs

Our approach to parallelising the solving of constraint problems has been previously described in [41]. This paper updates the description and, crucially, reports results from an application of the framework.

Constraint problems are typically solved by searching through the possible assignments of values to variables. After each such assignment, inference can rule out possible future assignments based on past assignments and the constraints. This process builds a search tree that explores the space of possible (partial) solutions to the constraint problem.

There are two different ways to build up these search trees – n-way branching and 2-way branching. This refers to the number of new branches which are explored after each node. In two-way branching, the left branch is of the form $x = y$ where x is a variable and y is a value from its domain. The right branch is of the form $x \neq y$.

The more commonly used way is 2-way branching, implemented for example in the Minion constraint solver [42], available at `http://minion.sf.net`. However, regardless of the way the branching is done, exploring the branches can be done concurrently. No information between the branches needs to be exchanged in order to find a solution to the problem.

We exploit this fact by, given the model of a constraint problem, generating new models which partition the remaining search space. These models can then be solved independently. We furthermore represent the state of the search by adding additional constraints such that the splitting of the model can occur at any point during search. The new models can be resumed, taking advantage of both the splitting of the search space and the search already performed.

A. Model splitting

Our new approach to the distributed solving of constraint problems requires the constraint solver to modify the constraint model but does not require explicit parallelisation support in the solver.

To split the remaining search space of a constraint problem, we signal the solver to stop. Now we partition the domain for the variable currently under consideration into n pieces of roughly equal size. Then we create n new models and to each in turn add constraints ruling out the other $n-1$ partitions of that domain. Each one of these models restricts the possible assignments to the current variable to one nth of its domain.

As an example, consider the case $n = 2$. The variable under consideration is x and its domain is $\{1, 2, 3, 4\}$. We generate 2 new models. One of them has the constraint $x \leq 2$ added and the other one $x \geq 3$. Thus, solving the first model will try the values 1 and 2 for x, whereas the second model will try 3 and 4.

The main problem when splitting constraint problems into parts that can be solved in parallel is that the size of the remaining search space for each of the splits is impossible to

predict reliably. This directly affects the effectiveness of the splitting however: if the search space is distributed unevenly, some of the workers will be idle while the others do most of the work.

Our approach allows to repeatedly split the search space after search has started. We use the procedure described above several times, each time adding more constraints to the model. In addition, we add *restart nogoods*, that is, additional constraints that tell the solver how much of the search space has been explored. Constraints added in a previous iteration are not affected by constraints added later – regardless of how often we split, no parts of the search space will be "lost", potentially missing solutions, and no part of the search space will be visited more than once.

Assume for example that we are doing 2-way branching (so that search branching decisions are simply that a variable does or does not take a specific value), the variable currently under consideration is again x with domain $\{1, 2, 3, 4\}$ and the branches that we have taken thus far are $x \neq 1$ and $x \neq 2$. The new generated models will all have the constraints $x \neq 1$ and $x \neq 2$ to get to the point in the search tree where we split the problem. Then we add constraints to partition the search space based on the remaining values in the domain of x similar to the previous example. The splitting process and subsequent parallel search is illustrated in Figure 1.

Using this technique, we can create new chunks of work whenever a worker becomes idle by simply asking one of the busy workers to split the search space. The search is then resumed from where it was stopped and the remaining search space is explored in parallel by the two workers. Note that there is a runtime overhead involved with stopping and resuming search because the constraints which enable resumption must be taken into account and the solver needs to explore a small number of search nodes to get to the point where it was stopped before. There is also a memory overhead because the additional constraints need to be stored.

We have implemented this approach in a recent version of Minion which has been released to the public. Experiments show that the overhead of stopping, splitting and resuming is not significant for large problems.

In practice, we run Minion for a specified amount of time before timing out, splitting based on the current state and resuming instead of splitting at the beginning and when workers become idle. This approach is much simpler and works well for large problems. The algorithm is detailed in Procedure 1. It creates an n-ary split tree of models for n new models generated at each split. The procedure for finding all solutions is similar. Initially, the potential for distribution is small but grows exponentially as more and more search is performed. We have found that $n = 2$ works well in practice because it is the easiest to implement and minimises the number of models created.

Minion models are stored in ordinary files. Each time the search space is split, two new input files are written. We modified the output produced by Minion to include the names of the files it produced and included the name of the file that was run when the search space was split in the new model files. This way, we can easily trace the splitting of the search space across the different files.

Input : constraint problem X, allotted time T_{max} and splitting factor $n \geq 2$
Output: a solution to X or "no solution" if no solution has been found

run the constraint solver with input X until termination or T_{max};
if `solved?(`X`)` **then**
  terminate workers;
  return *solution*;
else if `search space exhausted?` **then**
  return *"no solution"*;
else
  $X' \leftarrow X$ with new constraints ruling out search already performed;
  split X' into n parts $X'_1, \ldots, X'_n$;
  for $i \leftarrow 1$ **to** n **do in parallel**
    distSolve(X'_n, T_{max},n);
  end
end

Procedure distSolve(X,T_{max},n): Recursive procedure to find the first solution to a constraint problem distributed across several workers.

B. Comparison to existing approaches

The main advantages of our approach are as follows.

- We require only minimal modifications to existing constraint solvers. In particular, we do not require network communication and work stealing to be implemented.
- We do not require communication between workers to achieve good utilisation.
- The creation of separate model files when splitting increases the robustness against worker failure and provides accountability for every step.

For the purposes of a framework for solving large Artificial Intelligence search problems, the last point is especially crucial. The nature of the applications that we have in mind is such that it will be neither easy to verify whether a solution is valid nor feasible to repeat the calculations to get a confirmation. Furthermore, we have to be able to rely on the capability to recover from failures without having to repeat all the work.

By creating regular "snapshots" of the search done, the resilience against failure increases. This is in contrast to most other approaches, where the reliability of the system is decreased by using techniques that distribute work and rely on several machines instead of just a single one. Such systems have then to take additional measures to mitigate the problems caused by failures of machines or communication links. Every time we split the search space, the modified models are saved. As they contain constraints that rule out the search already

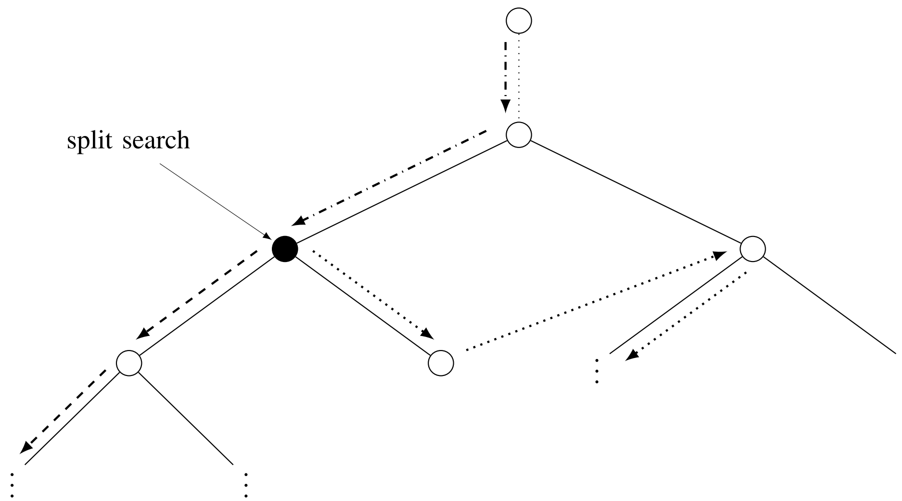

Fig. 1. Illustration of how search proceeds with splitting. The dashed and dotted line shows the search up to the solid black node where the models are split. The nodes subsequently explored by the two parallel searches are shown with dashed and dotted lines, respectively.

done, we only lose the work done after that point if a worker fails. This means that the maximum amount of work we lose in case of a total failure of all workers is the allotted time T_{max} times the number of workers $|w|$.

We note that our approach provides many of the advantages of efforts dedicated to improving the robustness and accountability of computations, e.g. [43], but is easier to implement and requires a minimal amount of supporting infrastructure.

Another consequence of our approach is that the solving process can be moved to a different set of workers after it has been started without losing any work. This may become necessary if parts of the problem require much more memory to solve than other parts. Instead of provisioning workers with a large number of resources for the entire duration of the computation, it becomes feasible to do this on-demand. This allows for excellent and easy integration with existing services that offer on-demand computing, such as a cloud.

C. Large-scale distribution

In the previous sections, we have described the techniques that enable the distribution of the solving of a constraint problem across a set of workers, but not the systems to take care of the actual distribution. The implementation of such systems is difficult, hence we decided to leverage tried-and-tested existing frameworks.

1) Condor: For the purposes of a framework that allows to distribute problems across a large number of heterogeneous workers, the Condor HPC system [44] is particularly suitable. It runs in many different operating and network environments and provides most of the functionality we require out of the box. In particular, it allows for the transfer of files that are created on the worker back to the master: the constraint models that split the search space.

Condor allows work units to be submitted to a central node which puts them in a queue to be executed when a worker becomes available. In our case, a constraint model is a unit of work and splitting the search space on one of the workers creates two new units of work that are transferred back to the master and queued for execution. The Condor job submission system makes sure that a job is executed to completion, i.e. if a worker node fails while it is processing a work unit, Condor requeues the work unit and sends it to a different worker.

Each Condor work unit needs to be created separately. In order to submit models that split the search space and are created during search, we have implemented a custom control system that monitors Condor and takes the appropriate action when split models are returned. The control system is an almost trivial piece of software that was very easy to implement – all of the heavy lifting is done by Condor.

While Condor is an adequate system for our needs, its installation is not always straightforward, meaning it can only be performed by trained personnel. This inherently limits usable machines to those which are accessible by such a user, and introduces a time factor to the addition of new machines to a cluster. Thus, any such cluster is strictly limited in its size by practicality, while the scale of problems we are aiming for might well ultimately require thousands, or tens of thousands, of machines. An institution would be hard-pressed to produce sufficient resources to make this available for a single project. Fortunately, the rise of the internet has facilitated so-called volunteer computing, where interested users can “donate” compute time to a project of their choice.

2) BOINC: The best-known framework for such projects is the Berkeley Open Infrastructure for Network Computing (BOINC) [45]. It has been used for many applications, including astrophysics [46], biology [47] and mathematics [48], focusing on the reliable distribution of individual small-scale “workunits” on fully heterogeneous nodes throughout the Web. Using its framework, we have produced a system along similar lines to the Condor one, allowing our central server to manage the distribution of jobs to any clients who request such work.

This system retains the total accuracy of the Condor or

single-node executions by actively mitigating node failure. In the case of loss of a client machine – a common event when working with many non-dedicated systems – the server will, like Condor, simply pass the job on to another worker, losing only limited time. Even in case of a total server collapse, the reinstating of a backup will restore complete server state, including the ongoing progress of partially executed tasks.

Other than robustness, the BOINC system has a number of advantages over single-node processing and even over Condor-style distribution, most of which relate to scalability. The nature of volunteer computing allows theoretically unlimited machines to contribute to any given task, with a bare minimum of technical knowledge required by the users. This flexibility provides inherent challenges such as the possibility of client falsification of data and potentially significant overheads involved in communication and readiness polling. There is also the practical issue of volunteer recruitment and retention. However, in practice these downsides are mitigated and even benefited from: communication overheads are reliably insignificant compared to the overally speedup gained through parallelisation. There have been many attempts to mitigate reliability and trust issues with clients, eg. [49], but we have chosen a simple redundancy framework, which suits our problem well given the fact that any anomalies can be detected through minor discrepancies in the final results.

IV. Application and discussion

The BOINC framework has been applied to a number of tasks in the area of the Progressive Party Problem. During these experiments, results were obtained which demonstrated the expected significant speed gains, as a single-node system running Minion locally. Despite various client problems, including machines shutting down mid-computation and users aborting tasks, results were returned that exactly matched those of a single node running Minion without any framework, to the total degree of accuracy that the problem required.

We have also had significant success with the evaluation of our Condor framework. This was done empirically by using it to compute the number of semigroups of order 9, a problem that had previously been solved using non-distributed search. We were able to confirm the known result on a number of different hardware configurations and splitting parameters, e.g. the time search is run before splitting the model.

Encouraged by the results of these experiments, we started the calculation of the number of semigroups of order 10. The hardware configuration throughout the computations varied, but the principal resources we used are shown in Figure 2. Here, one of the main advantages of our framework became apparent. The different resources we used were located in different networks that did not always have unrestricted connectivity to the other nodes. One of the research group clusters for example was behind a NAT in its own private network and unable to receive connections from outside this network. We were still able to utilise the resources to their full extent.

The submit machine and the Condor master shown in Figure 2 were not used for any of the computations, but only for the management of the calculations. It should be noted that there is no reason to have dedicated machines for those purposes as the resource requirements for the tasks they performed were very low. In principle, a machine used for management of the computations could also be used to perform computations itself.

The maximum number of processors that we used in parallel at any one time was about 150. One of the reasons for using the Amazon cloud was that it turned out that the machines we had available locally did not have enough memory to explore some parts of the search space efficiently. We were able to move those calculations to virtual machines in the Amazon cloud with suitable specifications and seamlessly integrate the results of those computations with the rest.

The total CPU time we expended to solve the problem (i.e. find exactly 256,587,290,511,904 semigroups from 10^{100} potential tables) was approximately 133 years. This effort was achieved in approximately 18 months; full details of the mathematics and the case-splits used are described in [23]. The limiting factor was the resources that were available to us. Even though we did not start with a short search time before splitting, enough split models to utilise all our resources were available after a few hours. For shorter computations, it might be desirable to facilitate faster splitting at the beginning to achieve good utilisation earlier, but for our purposes the framework as described previously was sufficient. The number of split models produced suggested that we could have utilised up to several thousand processors to a very high degree.

The robustness of our framework proved useful several times during the computations. Events that we successfully coped with included power and network outages, air-conditioning failures, physical machines being switched off and virtual machines disappearing. The damage in terms of computational effort lost was very limited in all cases. Condor was able to recover from most of these failures without any manual intervention by simply re-queueing the failed jobs. The verification of the distribution process revealed that because of the re-queueing a small part of the search space had been explored several times, but we were able to isolate and discard the duplicate model and output files.

After the computations finished, we were able to verify each step of the distribution and solving process. Therefore, we are confident that the result we obtained is correct. Ultimately, certainty of the correctness can only be established by either a new mathematical model that allows to calculate the computed number directly, or by independent verification through a second computation.

V. Conclusions and future work

We have presented two frameworks for the large-scale distribution of AI search in constraint programming across resources with minimal network connectivity requirements. Using two distribution mechanisms for classes of problems with as many as 10^{15} solutions, the framework has fully satisfied the joint requirements of scalability, robustness and verifiability. The framework is capable of scaling almost

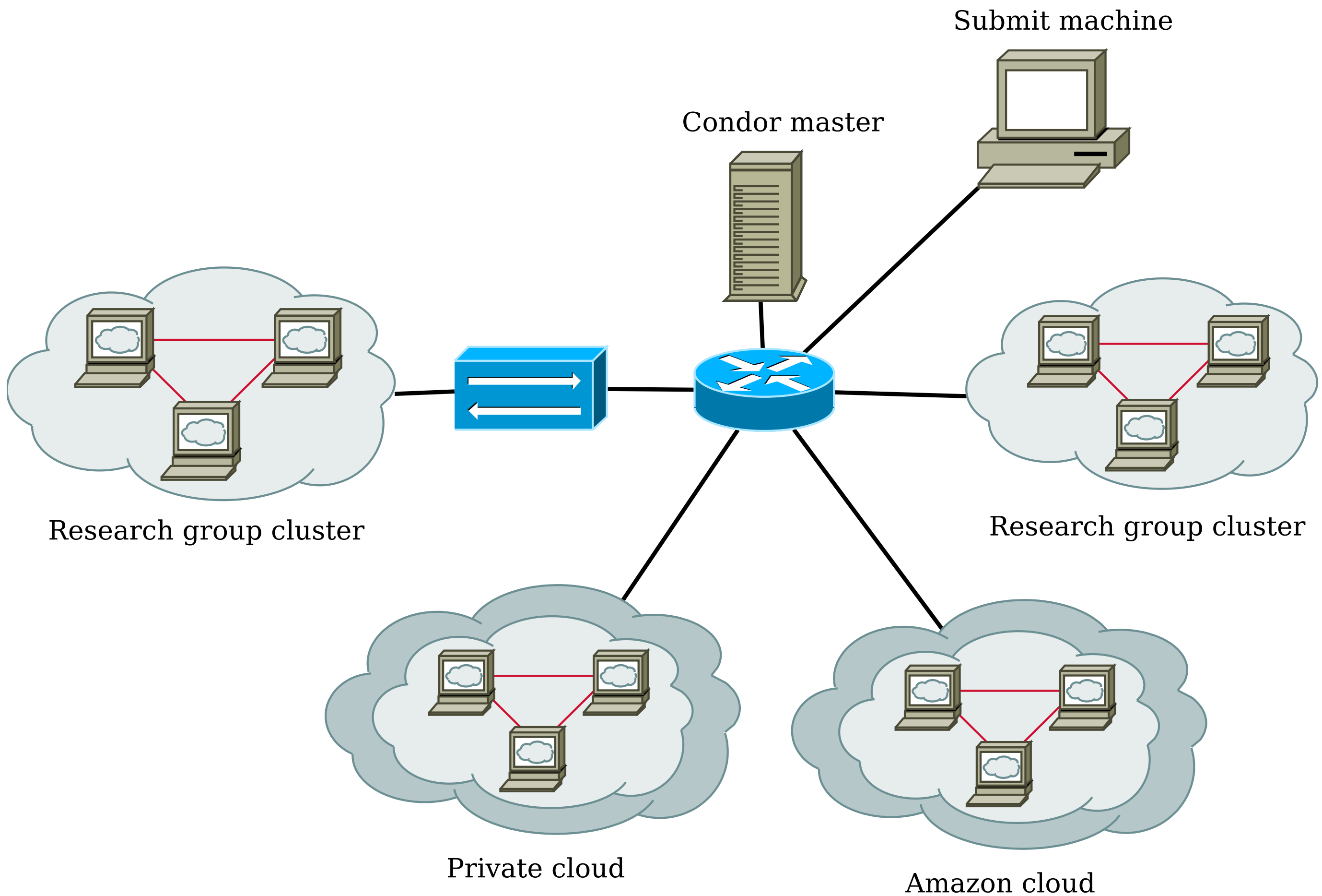

Fig. 2. Overview of resources used for the Condor-based enumeration of semigroups. We used two research clusters: a private cloud hosted in St Andrews and the Amazon cloud. One of the research clusters was behind a NAT switch such that no machines on the outside could reach it directly and all connections had to be initiated from within it.

seamlessly to a large number of distributed and heterogeneous resources while eliminating losses due to hardware failures or other client problems.

The type of our applications is unusual in the area of large-scale parallel algorithms. Instead of large amounts of data to process, we have a concise problem specification that takes vast computational resources to solve and must produce an utterly exact, reliable result. We believe (i) that the nature of such problems presents unique challenges to distribution which have rarely been considered thus far, and (ii) that the framework we have designed and the reasoning behind it will be foundations on which the e-Science community can build.

With that in mind, there is no indication that the positive experiences we have had with the specific applications described here is limited to the particular problems we investigated. We have, neither in the design of the framework nor the application problems, made any assumptions to that effect.

A theoretical comparison of the two distribution systems shows similarities in almost all important aspects, with the exception that the BOINC framework has fewer scale limitations. An obvious avenue for future work – apart from the application to new problems – is the empirical verification of this conjecture. A future application to the same problem would allow us to not only judge the differences in terms of distribution effectiveness and utilisation, but also to independently verify the results that we have obtained. While we are confident that we would indeed obtain the same result, an empirical verification would eliminate any doubts about this aspect of the framework.

We intend to release all components of the frameworks that are not already available to the public, thus enabling other researchers to tackle similarly large problems and providing a framework that we hope will prove useful to the research community.

Acknowledgments

The authors thank Chris Jefferson and Bilal Syed Hussein for useful discussions. Tom Kelsey is supported by UK EPSRC grant EP/H004092/1. Lars Kotthoff is supported by EU FP7 grants 288147 and 284715. Parts of the computational resources for this project were provided by an Amazon Web Services research grant. We thank the School of Computer Science, the Centre for Interdisciplinary Research in Computational Algebra and Cloud Co-laboratory (all of the University of St Andrews) for providing additional computational resources.

References

[1] G. Ingersoll, "Introducing apache mahout: Scalable, commercial-friendly machine learning for building intelligent applications," 2009.

[2] Y. Gil, E. Deelman, J. Blythe, C. Kesselman, and H. Tangmunarunkit, "Artificial intelligence and grids: Workflow planning and beyond," *IEEE Intelligent Systems*, vol. 19, pp. 26–33, 2004.

[3] M. Mann, C. Smith, M. Rabbath, M. Edwards, S. Will, and R. Backofen, "CPSP-web-tools: a server for 3D lattice protein studies," *Bioinformatics*, vol. 25, no. 5, pp. 676–7, 2009.

[4] T. Kelsey and L. Kotthoff, "The exact closest string problem as a constraint satisfaction problem," *CoRR*, vol. abs/1005.0089, 2010.

[5] D. J. Clancy and B. Kuipers, "Qualitative simulation as a temporally-extended constraint satisfaction problem," in *AAAI*, J. Mostow and C. Rich, Eds. The MIT Press, 1998, pp. 240–247.

[6] M. T. Escrig, L. M. Cabedo, J. Pacheco, and F. Toledo, "Several Models on Qualitative Motion as instances of the CSP," *Revista Iberoamericana de Inteligencia Artificial*, vol. 6, no. 17, pp. 55–71, 2002.

[7] N. Radke-Sharpe and K. White, "The role of qualitative knowledge in the formulation of compartmental models," *IEEE Transactions on Systems, Man and Cybernetics, Part C (Applications and Reviews)*, vol. 28, no. 2, pp. 272–275, May 1998.

[8] T. Menzies and P. Compton, "Applications of abduction: hypothesis testing of neuroendocrinological qualitative compartmental models." *Artificial intelligence in medicine*, vol. 10, no. 2, pp. 145–75, Jun. 1997.

[9] T. Kelsey and S. Linton, "Qualitative models of cell dynamics as constraint satisfaction problems," in *Proceedings of WCB12 – Workshop on Constraint Based Methods for Bioinformatics*, R. Backhoven and S. Will, Eds., 2012, pp. 16–23.

[10] J. Gao, Q. Yao, C. Harrison Bollinger, and D. XU, "Analysis and prediction of protein posttranslational modification sites," in *Algorithmic and Artificial Intelligence Methods for Protein Bioinformatics*, Y. Pan, J. Wang, and M. Li, Eds. John Wiley and Sons, Inc., 2013, pp. 91–106.

[11] T. Kelsey, L. Kotthoff, C. Jefferson, S. Linton, I. Miguel, P. Nightingale, and I. Gent, "Qualitative modelling via constraint programming," *Constraints*, vol. 19, no. 2, pp. 163–173, 2014.

[12] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft, "Recovery oriented computing (ROC): Motivation, definition, techniques,," University of California at Berkeley, Berkeley, CA, USA, Tech. Rep., 2002.

[13] G. Candea, A. B. Brown, A. Fox, and D. Patterson, "Recovery-oriented computing: Building multitier dependability," *Computer*, vol. 37, pp. 60–67, 2004.

[14] G. Goth, "Ultralarge systems: Redefining software engineering?" *IEEE Software*, vol. 25, pp. 91–94, 2008.

[15] M. Wallace, "Practical applications of constraint programming," *Constraints*, vol. 1, pp. 139–168, 1996.

[16] F. Rossi, P. van Beek, and T. Walsh, Eds., *The Handbook of Constraint Programming*. Elsevier, 2006.

[17] A. Distler and J. D. Mitchell, "The number of nilpotent semigroups of degree 3," *The Electronic Journal of Combinatorics*, vol. 19, no. 2, 2012.

[18] A. Distler and T. Kelsey, "The monoids of order eight and nine," in *Intelligent Computer Mathematics*. Springer Berlin Heidelberg, 2008, pp. 61–76.

[19] ——, "The monoids of orders eight, nine & ten," *Ann. Math. Artif. Intell.*, vol. 56, no. 1, pp. 3–21, 2009.

[20] Andreas Distler and Tom Kelsey, "The semigroups of order 9 and their automorphism groups," *Semigroup Forum*, pp. 1–20, 2013.

[21] T. Kelsey, S. Linton, and C. M. Roney-Dougal, "New developments in symmetry breaking in search using computational group theory," in *AISC*, ser. Lecture Notes in Computer Science, B. Buchberger and J. A. Campbell, Eds., vol. 3249. Springer, 2004, pp. 199–210.

[22] I. P. Gent, T. Kelsey, S. A. Linton, J. Pearson, and C. M. Roney-Dougal, "Groupoids and conditional symmetry," in *Principles and Practice of Constraint Programming–CP 2007*. Springer Berlin Heidelberg, 2007, pp. 823–830.

[23] A. Distler, C. Jefferson, T. Kelsey, and L. Kotthoff, "The semigroups of order 10," in *18th International Conference on Principles and Practice of Constraint Programming*, ser. Lecture Notes in Computer Science, M. Milano, Ed. Springer Berlin / Heidelberg, 2012, vol. 7514, pp. 883–899.

[24] T. Walsh, "CSPLib problem 013: Progressive party problem," http://csplib.org/Problems/prob013.

[25] I. P. Gent, B. S. Hussain, , C. A. Jefferson, L. Kotthoff, I. Miguel, G. F. Nightingale, and P. Nightingale, "Discriminating instance generation for automated constraint model selection," in *CP 2014*, 2014.

[26] V. N. Rao and V. Kumar, "Parallel depth first search. Part I. implementation," *Int. J. Parallel Program.*, vol. 16, no. 6, pp. 479–499, 1987.

[27] V. Kumar and V. N. Rao, "Parallel depth first search. Part II. analysis," *Int. J. Parallel Program.*, vol. 16, no. 6, pp. 501–519, 1987.

[28] Z. Collin, R. Dechter, and S. Katz, "On the feasibility of distributed constraint satisfaction," in *IJCAI*, 1991, pp. 318–324.

[29] V. N. Rao and V. Kumar, "On the efficiency of parallel backtracking," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 4, pp. 427–437, 1993.

[30] P. Sanders, "Better algorithms for parallel backtracking," in *Workshop on Algorithms for Irregularly Structured Problems*, 1995, pp. 333–347.

[31] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara, "Distributed constraint satisfaction for formalizing distributed problem solving," in *12th IEEE International Conference on Distributed Computing Systems*, 1992, pp. 614–621.

[32] M. Yokoo, E. H. Durfee, Ishida, and K. Kuwabara, "The distributed constraint satisfaction problem: Formalization and algorithms," *IEEE Trans. on Knowl. and Data Eng.*, vol. 10, no. 5, pp. 673–685, 1998.

[33] C. Schulte, "Parallel search made simple," in *Proceedings of TRICS*, 2000, pp. 41–57.

[34] L. Michel, A. See, and P. van Hentenryck, "Parallelizing constraint programs transparently," in *CP*, 2007, pp. 514–528.

[35] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, 1999.

[36] C. C. Rolf and K. Kuchcinski, "Load-balancing methods for parallel and distributed constraint solving," in *CLUSTER*, 2008, pp. 304–309.

[37] A. Petcu, "FRODO: a FRamework for Open/Distributed constraint optimization," Swiss Federal Institute of Technology (EPFL), Technical Report No. 2006/001, 2006, http://liawww.epfl.ch/frodo/.

[38] R. Ezzahir, C. Bessiere, M. Belaissaoui, H. Bouyakhf, U. Mohammed, and V. Agdal, *DisChoco: A platform for distributed constraint programming*, 2007.

[39] Y. Hamadi, *Disolver 3.0: the Distributed Constraint Solver version 3.0*, 2007.

[40] L. Michel and P. van Hentenryck, "A decomposition-based implementation of search strategies," *ACM Trans. Comput. Logic*, vol. 5, no. 2, pp. 351–383, 2004.

[41] L. Kotthoff and N. C. Moore, "Distributed solving through model splitting," in *3rd Workshop on Techniques for implementing Constraint Programming Systems (TRICS)*, 2010, pp. 26–34.

[42] I. P. Gent, C. Jefferson, and I. Miguel, "Minion: A fast scalable constraint solver," in *ECAI*, ser. Frontiers in Artificial Intelligence and Applications, G. Brewka, S. Coradeschi, A. Perini, and P. Traverso, Eds., vol. 141. IOS Press, 2006, pp. 98–102.

[43] A. Benabdelkader, M. Santcroos, S. Madougou, A. H. C. van Kampen, and S. D. Olabarriaga, "A provenance approach to trace scientific experiments on a grid infrastructure," in *IEEE 7th International Conference on e-Science*, Dec. 2011, pp. 134–141.

[44] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: The Condor experience," *Concurrency – Practice and Experience*, vol. 17, no. 2-4, pp. 323–356, 2005.

[45] D. P. Anderson, "BOINC: a system for Public-Resource computing and storage," in *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, ser. GRID '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 4–10.

[46] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "Seti@home: An experiment in public-resource computing," *Commun. ACM*, vol. 45, no. 11, pp. 56–61, Nov. 2002. [Online]. Available: http://doi.acm.org/10.1145/581571.581573

[47] M. Taufer, C. An, A. Kerstens, and C. L. Brooks III, "Predictor@ home: a" protein structure prediction supercomputer" based on public-resource computing," in *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*. IEEE, 2005, pp. 8–pp.

[48] M. Black and G. Bard, "Sat over boinc: An application-independent volunteer grid project," in *Proceedings of the 2011 IEEE/ACM 12th International Conference on Grid Computing*, ser. GRID '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 226–227. [Online]. Available: http://dx.doi.org/10.1109/Grid.2011.40

[49] T. Estrada, D. Flores, M. Taufer, P. Teller, A. Kerstens, and D. Anderson, "The effectiveness of threshold-based scheduling policies in boinc projects," in *e-Science and Grid Computing, 2006. e-Science '06. Second IEEE International Conference on*, Dec 2006, pp. 88–88.