The Design and Implementation of a Compositional Competition-Cooperation Parallel ATP System

Geoff Sutcliffe
Department of Computer Science, University of Miami, USA
Email: geoff@cs.miami.edu

Abstract

Key concerns in the development of more powerful ATP systems are to provide *breadth* of coverage – an ability to solve a large range of problems, and to provide greater *depth* of coverage – an ability to solve more difficult problems, within the same resource limits. This work describes the design and implementation of CSSCPA, a compositional competition-cooperation parallel ATP System. CSSCPA combines existing high performance ATP systems in a framework that allows them to work independently, but also allows communication of intermediate results. The performance data shows that CSSPCA has high breadth and depth of coverage.

1 Introduction

Automated Theorem Proving (ATP) is concerned with the development and use of systems that automate sound reasoning: the derivation of conclusions that follow inevitably from facts. Current ATP systems are capable of solving non-trivial problems, e.g., EQP [McC00a] solved the Robbins problem [McC97]. However, the search complexity of most interesting problems is enormous, which has two consequences for ATP. First, in order to solve certain types of hard problems, it is typically necessary to tune an ATP system for the problems. Such tuning almost inevitably has the consequence that the system can no longer solve some other problems, i.e., gain in one direction is at the cost of loss in another.[1] Second, there are many problems that still cannot currently be solved within realistic resource limits. Therefore, key concerns in the development of more powerful ATP systems are to provide *breadth* of coverage – an ability to

[1] It would be marvelous if the characteristics of ATP problems were sufficient to correctly identify every type of problem for which tuning has resulted in successful solution, for then the appropriately tuned features of an ATP system could be automatically invoked when the type of problem is recognized. However, thus far such recognition seems impossible.

solve a large range of problems, and to provide greater *depth* of coverage – an ability to solve more difficult problems, within the same resource limits.

One approach to providing breadth of coverage is the development of *compositional* ATP systems. Compositional systems are built from multiple component systems, and use one or more of the components when attempting to solve a problem. Compositional systems may be characterized by the way in which the components are run: *Time slicing* systems select one or more components, allocate some fraction of the available CPU time to each of the selected components, and then run the components one after the other until a solution is found. Examples of time slicing systems are Gandalf [Tam97] and more recent versions of Vampire [RV99]. *Competition* systems similarly select components and allocate CPU time, but then run the components in parallel (or at least concurrently, according to the number of CPUs available) until a solution is found. Examples of competition systems are RCTHEO [Ert92] and SSCPA [SS99].

Both time slicing and competition systems rely on the phenomenon that components can be selected so that there is a significant difference in the set of problems that each can solve quickly – the properties of sub-linearity and complementarity [SW99]. Time slicing systems have the advantage that they are inherently well suited to single CPU machines, and the components can be given 'dedicated' access to the CPU in the order of likelihood of solving the problem. Competition systems have the advantage that they can take advantage of multiple CPU architectures (especially SMP machines), and there is no need to decide which components are more likely to solve the problem. For both time slicing and competition systems, greater diversity across the components provides greater breadth of coverage.

The components of a simple compositional system do not cooperate, with no communication of control information or intermediate results. Such compositional systems can solve at most the union of the problems that the individual components can solve within the same total CPU time limit, i.e., there is no gain in depth of coverage (and some problems that can be solved by individual components within the full CPU time limit may not be solved within the fraction of the CPU time limit allocated to the components in the compositional setting). The capabilities of a compositional system can be significantly affected by the addition of cooperation. The communication of control information is problematic if the components are diverse (as recommended above), because they have different search spaces and the control information from one component is typically inappropriate for another. One coarse grained way of effecting the communication of control information is to have components with different control strategies, and to start and stop the components according to evidence of their success in the context of the overall system. This approach is taken in the DISCOUNT system [ADF95].

The communication of intermediate results is significantly easier. For time slicing systems, the intermediate results generated by an unsuccessful component in the sequence are passed on to subsequent components. Examples of time-slicing-cooperation systems are Gandalf [Tam97] and the e-iterator strategy within E-SETHEO [SW99]. For competition systems, a protocol has to be

established to allow communication of intermediate results during runtime. For competition systems in particular, communication between diverse component systems often has synergistic effects, leading to "super linear" speed ups. This is due to cross fertilization between the components, as a component may receive useful intermediate results that it would not generate itself. Examples of competition-cooperation systems are HPDS [Sut92] and TECHS [DF99]. The addition of cooperation to a compositional system is an effective way of increasing the depth of coverage of the system. The synergistic effects allow the system to solve problems that none of the component systems are able to solve independently. As before, diversity is important, as this provides more extreme cross fertilization.

The above survey suggests that a *C*ompetition-*C*ooperation *C*ompositional system, running genuinely in *P*arallel on a multi-CPU machine (a *CCCP* system), has high prospects for attaining both breadth and depth of coverage.[2] The following sections of this paper examine a particular instance of the design and implementation of a CCCP system. Section 2 looks at the issues and choices in the design of such a system. Sections 3 and 4 describe the design and implementation of the CSSCPA[3] system, highlighting the benefits of the design decisions made, and difficulties encountered. Section 5 provides performance data, and Section 6 concludes the paper.

2 Design Issues for a CCCP ATP System

There are three main issues that need to be addressed in the design of a CCCP system:

- How the components will be controlled and monitored. Aspects of this include how the components will be started, how the resource usage of the components will be allocated and limited, and how the components will be stopped when a solution is found or the resource limits are exceeded.
- What data formats will be used. The data formats for the input problem, the intermediate results that are communicated, and the output, all need to be considered.
- The granularity and mode of communication between the components. Options here range from fine grained point-to-point communication, where small intermediate results are transferred immediately and directly to other components, through to coarse grained bulk transfer of many intermediate results at widely spaced intervals.

The choices for each of these issues are constrained to a large degree by the nature of the component systems. There are three common types of component systems, each with quite different characteristics, which affect the range of options available in the design.

[2] Other approaches to using parallelism in ATP are surveyed in [SS94] and [Bon00].
[3] Pronounced "sea skipper".

The highest degree of design flexibility is available when the components are designed and developed in-house. In this case there is access to and understanding of the design and implementation of the components, and it is easily possible to make adaptations in the components specifically for the CCCP system. From the control perspective, it is possible to directly manage the component processes, and to have the components internally limit their resource usage. A single data format can be designed and used consistently for input, communication, and output. Both fine and coarse grained communication of intermediate results are possible. In particular, if a common data format is used, there is no overhead of format conversion, which is particularly attractive if the communication between components is fine grained. The drawback of using in-house components is that significant effort has to be expended in order to design and implement components that have sufficiently high performance. In many cases there is simply not enough expertise and programmer-power to achieve this. HPDS is an example of a CCCP system that benefited from the advantages and also suffered from the disadvantages of this approach.

An intermediate degree of design flexibility is available when using high performance components developed elsewhere, but for which the source code is available and understandable. Such components can be modified as required to run in the CCCP system. Modifications can be made to make component control easily possible, internal translation of data formats can be implemented, and communication hooks can be inserted. The main advantage of this option is the adoption of existing high performance systems, which may have required significant effort and expertise to develop. However, the effort required to make the necessary modifications to someone else's code is often prohibitive. Further, as new versions of the components are released by their developers, it is necessary to port the modifications to the new versions. As a result, it is difficult to keep such a CCCP system upgraded to the most recent component technology. TECHS is an example of such a CCCP system.

The least design flexibility is available when using high performance components developed elsewhere, without any intention of making modifications. There are significant advantages and disadvantages of this approach. Controlling the execution and monitoring the CPU usage of a component may be difficult, especially if the component runs multiple processes. Although starting a component may be easy, it may then be difficult to monitor and limit its resource usage, or to stop all processes of the component.[4] The data formats of the components are likely to be incompatible, and it is necessary to implement external format translation for at least the input and communication data. Almost certainly the components will not accept intermediate results during runtime, thus making a coarse grained communication model necessary. These drawbacks require solutions at both the logical and practical levels. If solutions

[4]For example, under UNIX, if a process in the middle of a three level process hierarchy terminates, leaving the bottom level process to communicate with the top level process using files, the bottom process is no longer in the process hierarchy of the top process. It is then difficult to stop the bottom level process, and the CPU time of the bottom process is not accumulated as child CPU usage in the top level process.

can be found, however, there are some attractive aspects to this approach, all stemming from the fact that the components are used without modification. First, the highest performance components can be used, and their performance will not be degraded through modifications required to fit them into the CCCP system. Second, it is not necessary to have access to the components' sources. This makes it possible to use components that for some reason, e.g., proprietary constraints, can be distributed only in binary form. Third, as the CCCP system can be concerned with only the external presentation of the components, it is likely to be easy to replace a component by a newer version. This is because the external presentation of a (component) system often remains the same while internal structures are (possibly significantly) changed.

3 The Design of CSSCPA

CSSCPA is a CCCP system for problems in the CNF of first order classical logic, expressed in the TPTP CNF syntax [SS98]. CSSCPA uses existing high performance components, without any modification. The components must have an option to produce, on their standard output, intermediate clauses that are logical consequences of the input problem. In addition to the component ATP systems, CSSCPA employs a formula librarian (the FLi) that can do subsumption and also detect unit contradictions in the clauses it holds.

When given an ATP problem, CSSCPA first sends a copy of the problem to the FLi, where it is stored. CSSCPA then selects components to use, based on a database of information about the eligible components' strengths for various problem types. CSSCPA starts the components, and parses their standard outputs for logical consequences. Each logical consequence is forwarded to the FLi. If at any time a component finds a solution, then CSSCPA is stopped and success is reported.

The FLi keeps an incoming clause only if keeping it improves the overall quality of its clause set, in the sense that a better clause set is one that is easier to refute. For example, the FLi's clause set is improved if an incoming logical consequence subsumes a clause in the set (see Section 4 for further details of the clause set evaluation). Whenever the quality of clause set in the FLi improves, the FLi reports the improvement to CSSCPA. When the clause set in the FLi has improved significantly relative to the original input problem, CSSCPA stops the components. CSSCPA then collects the improved clause set from the FLi, and restarts with the improved clause set as the input problem.

When a proof is found, CSSCPA outputs the original problem file, the sequence of improved problem files, and the component system's proof. The clauses in the improved problem files are annotated to indicate their source, either from the original input problem, or from one of the component ATP systems. This provides sufficient information to construct a monolithic proof, which can be checked using standard techniques.

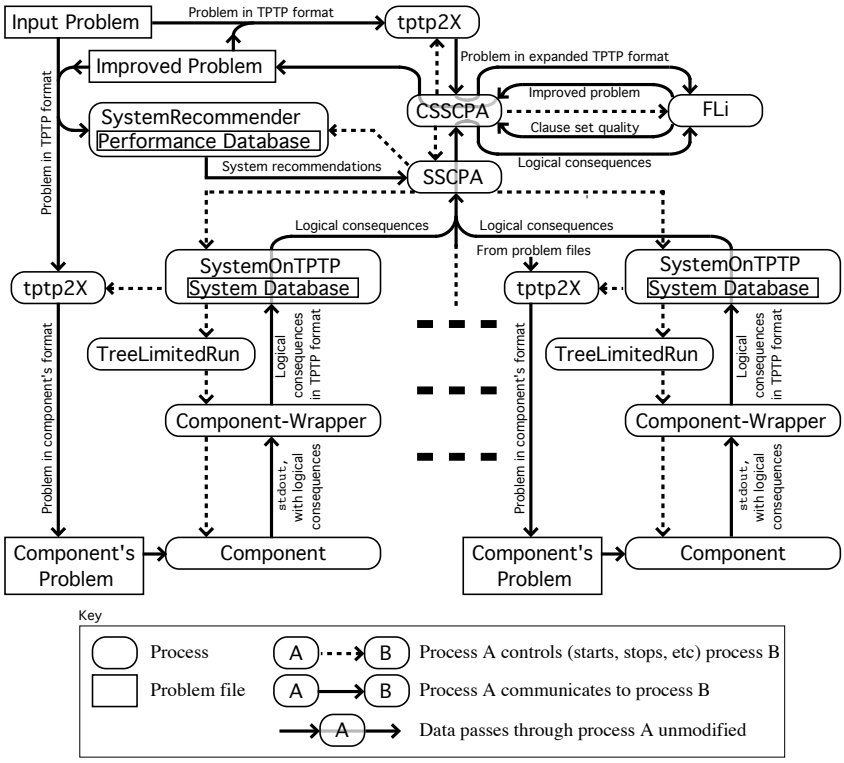
CSCCPA creates a sequence of successively easier problems to solve. This technique is called *iterative easing*. The arrangement is clearly sound, provided

that the component systems output only logical consequences. On a macro level CSSCPA may be viewed as a time-slicing compositional system, in which each CSSCPA iteration is one component system.

4 The Implementation of CSSCPA

The overall process architecture of the CSSCPA implementation is shown in Figure 1.

Figure 1: The CSSCPA Architecture



When given an ATP problem, CSSCPA uses the `tptp2X` utility to expand any `include` statements in the problem, and forwards the input problem clauses to the FLi. The FLi is an external module of the E ATP system [Sch01], and thus employs the efficient data structures and formula manipulation routines in the E implementation. When the FLi has received and stored all the input clauses, it reports the number of clauses, number of literals, and sum weight of the clause set, on its standard output. This information is captured by CSSCPA as the quality of the input problem clauses. The FLi then sits in a loop, reading clauses

(the logical consequences from the component ATP systems) on its standard input, processing them, and reporting any improvements of the clause set quality on its standard output. CSSCPA monitors the standard output of the FLi. If the reported quality of the FLI's clause set passes a threshold, CSSCPA stops the execution of the component systems and collects the improved clause set from the standard output of the FLi. If the clause set does not contain a unit contradiction, CSSCPA restarts using the improved clause set as the input problem.

The selection and execution of components for CSSCPA is done by the SSCPA system [SS99]. To select components, SSCPA runs a `SystemRecommender` program. The `SystemRecommender` accesses a database of information about the eligible components' strengths. The database is built from an evaluation of performance data for the ATP systems on TPTP problems [SS01]. The database assigns problems to one of 16 Specialist Problem Classes (SPCs) based on syntactic problem characteristics, and ranks the systems within each SPC. When a new problem is presented, its SPC is identified and the best performing systems for the SPC are then known. SSCPA divides the CPU time remaining equally between the selected systems. SSCPA then invokes SystemOnTPTP [Sut00] to run each of the selected components on the problem. Note that the set of selected components may change between iterations in CSSCPA, due to changes in the characteristics of the clause set.

Each instance of SystemOnTPTP accesses a database of information about the ATP systems to determine the format in which its ATP system requires the problem. The SystemOnTPTP then uses the `tptp2X` utility to do the necessary transformations and formatting.

SystemOnTPTP uses a control process (called `TreeLimitedRun`) to control and monitor its ATP system. The control process starts the ATP system, monitors the resource usage of the system, imposes resource usage limits, and has sufficient information to be able to stop all the processes that the system has running when a resource limit is exceeded (the CPU time allocated, a wall clock time limit, and a memory limit). The control program monitors the CPU usage of the ATP system's processes by scanning the `/proc` file system. The wall clock limit is implemented by an `alarm` system call within the control process. The memory limit is imposed through use of the `setrlimit` system call. When the CPU or wall clock time limit is reached, the control process scans the `/proc` file system for the system's processes, and uses a `kill` system call to stop them all.

The control process runs the specified ATP system inside a component wrapper. The wrapper scans the standard output of the ATP system and extracts clauses that are logical consequences of the input problem. The wrapper then translates the clauses to TPTP format before writing them to its standard output. The standard output of the wrapper is captured by the corresponding instance of SystemOnTPTP, which echoes the information to its standard output. SSCPA collates the standard outputs from the SystemOnTPTP instances, and writes them to its standard output. This is captured by CSSCPA, and the logical consequences are then forwarded to the standard input of the FLi.

In the current implementation of CSSCPA, the quality of the FLi's clause set

is measured in two ways: the total weight (symbol count) of the clause set, and the average clause weight. The quality of the clause set improves when either of these decreases. The total weight decreases when an incoming clause subsumes clauses whose sum weight is greater than that of the incoming clause. The average clause weight decreases when an incoming clause's weight is less than the current average clause weight. The clause set in the FLi is considered to have "improved significantly" when either of the quality measures goes down by some fraction of the input clause set's measures. The fractions are parameters to CSSCPA.

CSSCPA, SSCPA, SystemOnTPTP, and the component wrappers are all implemented in `perl`. The FLi and `TreeLimitedRun` are implemented in C. `tptp2X` is implemented in Prolog. The `perl` implementation of key components may be a bottleneck in the communication, but at this stage it seems to be acceptable. It is noteworthy that all inter-process communication uses the standard input and output streams. At the bottom level, this is necessary for capturing the logical consequences from the component ATP systems, given the commitment to using unmodified components. The decision to use standard IO streams for the other levels followed as a consequence.

5 Performance

Initial testing of CSSCPA has been done using the 1745 TPTP problems that are non-Horn, have some (but not only) equality literals, and have an infinite Herbrand universe (i.e., a very general class of problems). The same three components were selected all the time, they being SPASS 1.03 [WAB+99], E 0.62 [Sch01], and Otter 3.0.6 [McC00b], all running in their default "auto" modes (splitting was turned off in SPASS, so that only logical consequences were generated). A 300 second time limit was imposed, individually when testing the individual component systems, and as a total in the CSSCPA setting. Table 1 summarizes the results. The SSCPA column shows the results for the naive mode of SSCPA, in which the three components are run in competition parallel with a CPU time limit of 100 seconds for each component. The SSCPA* column shows the SSCPA results with a CPU time limit of 300 seconds for each component.

Table 1: CSSCPA Results

	CSSCPA	E	Otter	SPASS	SSCPA	SSCPA*
Solved by	686	616	364	630	673	723
CSSCPA, not by other	131	329	91	75	52	
other, not by CSSCPA	65	11	39	62	89	

CSSCPA solved 52 problems that none of the components solved within 300 seconds, and 25 `SWC` problems with a TPTP difficulty rating of 1.00, i.e., 25

problems that no existing ATP system is known to be able to solve. These results show that CSSCPA has high depth of coverage. CSSCPA solves more problems than any component, and 13 more problems than their composition in SSCPA. These results show that CSSCPA has high breath of coverage (although it would be even better if CSSCPA subumed (solved a superset of the problems solved by) the components and SSCPA).

It is interesting that there are 62 problems solved by SSCPA but not by CSSCPA. The essential differences between SSCPA and CSSCPA are the communication of intermediate results and a reduction of the CPU time limit on each component in successive CSSCPA iterations. Clearly the "improvements" in the problem and the reduced time limits affect the components' abilities to solve those problems.

It should be noted that CSSCPA's performance on a given problem can change from run to run, due to changes in the operating system's scheduling of the component ATP systems, which affects the order in which logical consequences are forwarded to the FLi.

6 Conclusion

The need for powerful ATP systems that have both breadth and depth of coverage has motivated the design and implementation of the compositional competition-cooperation parallel ATP system CSSCPA. CSSCPA combines existing high performance ATP systems in a framework that allows them to work independently, but also allows communication of intermediate results. The performance data shows that CSSPCA has high breadth and depth of coverage.

It is planned to extend the range of component systems available to CSSCPA. In particular, the use of analytic provers, e.g., model elimination or tableau based provers, with lemma generation capabilities, seems attractive. It is expected that there will be strong cross fertilization between saturation systems and analytic systems, due to their different deduction and search strategies.

The soundness of CSSCPA is dependent on the soundness of the components, and also (from a practical viewpoint) the correct capturing and forwarding of logical consequences. It is planned to independently verify CSSCPA proofs by converting the sequence of improved problems into a monolithic proof, and applying standard proof checking techniques.

Acknowledgement: Thanks to Stephan Schulz for implementing the FLi.

References

[ADF95] J. Avenhaus, J. Denzinger, and M. Fuchs. DISCOUNT: A System for Distributed Equational Deduction. In Hsiang J., editor, *Proceedings of the 6th International Conference on Rewriting Techniques and Applications*, pages 397–402. Springer-Verlag, 1995.

[Bon00] M.P. Bonacina. A Taxonomy of Parallel Strategies for Deduction. *Annals of Mathematics and Artificial Intelligence*, 29:223–257, 2000.

[DF99] J. Denzinger and D. Fuchs. Cooperation of Heterogeneous Provers. In T. Dean, editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence* , pages 10–15. Morgan Kaufmann, 1999.

[Ert92] W. Ertel. OR-Parallel Theorem Proving with Random Competition. In Voronkov A., editor, *Proceedings of the 1992 Conference on Logic Programming and Automated Reasoning* , number 624 in Lecture Notes in Artificial Intelligence, pages 226–237. Springer-Verlag, 1992.

[McC97] W.W. McCune. Solution of the Robbins Problem. *Journal of Automated Reasoning*, 19(3):263–276, 1997.

[McC00a] W.W. McCune. EQP: Equational Prover. http://www-unix.mcs.anl.gov/AR/eqp/, 2000.

[McC00b] W.W. McCune. Otter: An Automated Deduction System. http://www-unix.mcs.anl.gov/AR/otter/, 2000.

[RV99] A. Riazanov and A. Voronkov. Vampire. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, number 1632 in Lecture Notes in Artificial Intelligence, pages 292–296. Springer-Verlag, 1999.

[Sch01] S. Schulz. System Abstract: E 0.61. In R. Gore, A. Leitsch, and T. Nipkow, editors, *Proceedings of the International Joint Conference on Automated Reasoning*, number 2083 in Lecture Notes in Artificial Intelligence, pages 370–375. Springer-Verlag, 2001.

[SS94] C.B. Suttner and J. Schumann. Parallel Automated Theorem Proving. In L. Kanal, V. Kumar, H. Kitano, and C. Suttner, editors, *Parallel Processing for Artificial Intelligence 1*, pages 209–257. Elsevier Science, 1994.

[SS98] G. Sutcliffe and C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[SS99] G. Sutcliffe and D. Seyfang. Smart Selective Competition Parallelism ATP. In A. Kumar and I. Russell, editors, *Proceedings of the 12th Florida Artificial Intelligence Research Symposium*, pages 341–345. AAAI Press, 1999.

[SS01] G. Sutcliffe and C.B. Suttner. Evaluating General Purpose Automated Theorem Proving Systems. *Artificial Intelligence*, 131(1-2):39–54, 2001.

[Sut92] G. Sutcliffe. A Heterogeneous Parallel Deduction System. In R. Hasegawa and M.E. Stickel, editors, *Proceedings of the Workshop on Automated Deduction: Logic Programming and Parallel Computing Approaches, FGCS'92*, 1992.

[Sut00] G. Sutcliffe. SystemOnTPTP. In D. McAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction*, number 1831 in Lecture Notes in Artificial Intelligence, pages 406–410. Springer-Verlag, 2000.

[SW99] G. Stenz and A. Wolf. E-SETHEO: Design, Configuration and Use of a Parallel Automated Theorem Prover. In N. Foo, editor, *Proceedings of AI'99: The 12th Australian Joint Conference on Artificial Intelligence*, number 1747 in Lecture Notes in Artificial Intelligence, pages 231–243. Springer-Verlag, 1999.

[Tam97] T. Tammet. Gandalf. *Journal of Automated Reasoning*, 18(2):199–204, 1997.

[WAB+99] C. Weidenbach, B. Afshordel, U. Brahm, C. Cohrs, T. Engel, E. Keen, C. Theobalt, and D. Tpoic. System Description: SPASS Version 1.0.0. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, number 1632 in Lecture Notes in Artificial Intelligence, pages 378–382. Springer-Verlag, 1999.