A Heterogeneous Parallel Deduction System

Geoff Sutcliffe
Dep't of Computer Science, Edith Cowan University &
Dep't of Computer Science, The University of Western Australia
Perth, Western Australia geoff@cs.uwa.edu.au

Abstract
This paper describes the architecture, implementation and performance, of a heterogeneous parallel deduction system (HPDS). The HPDS uses multiple deduction components, each of which attempts to find a refutation of the same input set, but using different deduction formats. The components cooperate by distributing clauses they generate to other components. The HPDS has been implemented in Prolog-D-Linda. Prolog-D-Linda provides appropriate data transfer and synchronisation facilities for implementing parallel deduction systems. The performance of the HPDS has been investigated.

Parallel Deduction Systems

A parallel deduction system is one in which multiple deduction components run in parallel on separate processors. This is distinct from those deduction systems which run multiple deduction components alternately, such as the unit preference system [Wos, Carlson & Robinson G.A., 1964], and those which are only conceptually parallel systems. Parallel deduction systems can be categorised along three axes.

Homogeneous or Heterogeneous? In a homogeneous system the multiple deduction components all use the same deduction format. Homogeneous systems do not change the search space of the underlying deduction format. Rather they take advantage of added computing power to distribute the work of searching that space. An example of such a system is ROO [Lusk, Slaney & McCune, 1991]. In a heterogeneous system the multiple deduction components use different deduction formats. The advantage of heterogeneity is that each deduction component has a different search space. Further, it is necessary for only one of the components to use a complete deduction format. This feature permits incomplete formats, which are fast on average, to be used as part of complete parallel deduction system. An example of such a system is GLD||UR [Sutcliffe, 1991]. Between homogeneous and heterogeneous are the pseudo heterogeneous systems. The components of these systems all use the same deduction format, but each component has different set-up parameters. Each component will thus have a similar search space, but will search the space in a different manner. An example of such a system is Ertel's Random Competition system [Ertel, 1991].

Common or Separate input sets? To implement a common input set for multiple deduction components, some kind of shared memory architecture is necessary. On the other hand, if each component maintains its own copy of the input set then more common computer architectures can be used. ROO is an example of a system that uses a common input set, while the Random

Competition system is one that uses separate input sets. An approximation of a common input set can be implemented by using separate input sets which are updated in parallel. The approximation can be implemented provided there is some form of inter-component communication mechanism. This approach is taken in GLD||UR.

Homogeneous systems require a common input set, for otherwise the components will duplicate each other's search. In a heterogeneous system it is ok to have separate input sets, as the components do different things naturally. However, there is an advantage to having a common input set in a heterogeneous system. As each component has a different search space, the clauses created in each component may not be created in others. By making all clauses available to all components, 'cross fertilisation' is achieved. Although hard to quantify, this effect has the potential to significantly affect a parallel system's performance. Further, if the parallel update approach is used, the distribution of clauses can be controlled so that different components' sets are updated differently.

Synchronous or Asynchronous? An asynchronous system has the advantage that components do not have to wait for each other, and greater use is made of the available computing power. On the other hand, there may be advantage to be had in making the multiple components aware of and supportive of each others activities.

This paper introduces a heterogeneous parallel deduction system (HPDS), employing a chain format linear deduction component, a UR-deduction component, and a hyper-resolution component. The HPDS is the logical successor of GLD||UR. The components of the HPDS each maintain their own copy of the input set. Each input set is updated with clauses created locally and with clauses created in the other components. The components run asynchronously.

The Parallel Implementation Environment

A prerequisite to the implementation of a parallel deduction system is to decide upon and, if necessary, develop an appropriate parallel programming environment. The HPDS has been implemented using Prolog-D-Linda [Sutcliffe & Pinakis 1991]. Linda is a programming framework of language-independent operators which may be injected into existing programming languages, resulting in new parallel programming languages. Linda permits cooperation between parallel processes by controlling access to a shared data structure called a *tuple space*. Manipulation of a tuple space is only possible using Linda operators (`out`, `in` and `rd`). Parallel execution is provided by an operator (`eval`) which starts new processes. Generically, Prolog-Linda is the extension of Prolog that supports a tuple space and the Linda operators. Prolog-D-Linda is our implementation of Prolog-Linda. Prolog-D-Linda is built on top of SICStus Prolog. It runs on a network of SUN SPARC workstations running SUN OS 4.0.3, connected by Ethernet.

In Prolog-D-Linda the tuple space is distributed (hence "Prolog-Distributed-Linda"). The tuple space and associated operations are implemented in *server* processes. The distribution of tuples across the servers is determined by a user supplied Prolog program. Linda operations in *client* processes are translated into requests which are passed to the appropriate server. One server is designated the *eval-server*, and is responsible for starting client processes requested in `eval` operations. The servers are started by a *controller* process. After starting the servers, the controller works as the standard input and output device for all the servers, and for clients that are started via an `eval` request. This feature enables servers and clients, that are not associated with a terminal, to have user interaction.

The HPDS Architecture

The deduction components of the HPDS are Guided Linear Deduction (GLD) [Sutcliffe, 1992] (GLD is a chain format linear deduction system), a UR-deduction [Overbeek, McCharen & Wos, 1976] component, and a hyper-resolution [Robinson J.A., 1965] component. Each component maintains its own input set, asynchronously updating it with clauses created in the other components at a time convenient to itself. All of the components have been implemented in Prolog. Common features of the components are :

- All use the chain format for clauses. They all use the same code for their deduction operations, processing of clauses created, and manipulation of their input sets. This makes it easy to distribute the clauses that each creates.
- Each component creates clauses that can be used by the others. All the components apply back and forward subsumption to all clauses that they create. The subsumption checks are applied before the clauses are distributed to the other components. Information indicating which clauses are subsumed accompanies each clause that is distributed.
- Each component uses a consecutively bounded search as its overall search strategy. Although the value bounded is different in each component, the same implementation is used in each component. If a new clause is added to the input set in a search iteration, the the bound value is increased only minimally at the end of the iteration. Further, another iteration is always executed if a new clause has been added to the input set in the iteration. In the HPDS these features are affected not only by clauses created locally, but also by clauses received from the other components.

The GLD Component

GLD is based on Shostak's Graph Construction (GC) procedure [Shostak, 1976], but also incorporates features from other chain format linear deduction systems. In particular, GLD has a combined lemma/C-literal mechanism, which improves on the original separate mechanisms in Loveland's Model Elimination (ME) procedure [Loveland, 1969] and the GC procedure. The lemmas created by this mechanism are the clauses that are distributed to the hyper-resolution and UR components. GLD uses an extended unit preference strategy in all extension operations, and thus productively uses the unit clauses which it receives from the UR component. GLD does not accept clauses created in the hyper-resolution component, as it was found that the large number of clauses created by hyper-resolution caused an unacceptable increase in the size of the GLD search space.

The UR Component

The UR component's strength lies in its ability to solve many Horn problems very quickly. The UR component is a useful adjunct to the complete GLD and hyper-resolution components. The clauses created in the two complete components are used as nuclei in the UR component. This leads to the UR component reporting refutations for problems that it cannot solve independently.

The Hyper-resolution Component

The hyper-resolution component of the HPDS implements positive hyper-resolution. Non-factored hyper-resolvants are distributed to the other components. Like the GLD component, the hyper-resolution component benefits from using the unit clauses produced by the UR component. There is some possibility of an overlap between the clauses created in this component and those created in the GLD and UR components. However, GLD is a back chaining system, while hyper-resolution is forward chaining. Thus the searches of these two components are in the opposite direction. Thus each should benefit from the work that the other does at the other end of the general search space. The overlap with the UR component is dealt with by the subsumption checks.

The Combination

The completeness of the HPDS is assured by the completeness of GLD and hyper-resolution. The difference between these components running independently and running in the HPDS, is the addition of new input clauses created in the other components. Such clauses are subject to the same subsumption checks as those created locally. By having the consecutively bounded search run another iteration whenever any new clause is added to the input set, both the GLD component and the hyper-resolution component remain complete within the HPDS.

The HPDS Implementation

Each deduction component of the HPDS runs as a client process in Prolog-D-Linda. The deduction components do not communicate with each other directly, but rather via a deduction controller (also a Prolog-D-Linda client). The deduction controller is responsible for starting the deduction components, and for controlling the distribution of clauses created in the deduction components. Communication between the deduction components and the controller is via the Prolog-D-Linda tuple space. When a deduction component wishes to pass any data to the controller, it `outs` a tuple of the form controller(<source>,<type>,<data>). The <source> is the name of the deduction component which produced the tuple. The <type> field indicates the type of the <data>. There are four possible <type> values□: `clause`, `stop`, `io` and `statistics`. The controller retrieves these tuples from the tuple space using the `in` operation. Similarly, the controller passes data to the deduction components by placing tuples of the form <destination>(controller,<type>,<data>) into the tuple space.

When the deduction controller is started, it collects information about each of the deduction components from the user. This includes the name of component's source code file, the Prolog query required to start the component, and a flag indicating whether or not the component is complete. Each deduction component is then started using the `eval` operation. The task of the controller is then to continuously retrieve its message tuples from the tuple space and to deal with the data according to the value of the <type> field.

The deduction components, as well as executing their deductions appropriately, must check the tuple space at regular intervals for messages from the controller. Again, these messages are dealt with according to their <type>. The GLD component checks for messages after each centre clause is deduced. The UR component checks for messages each time a new nucleus is chosen. The hyper-resolution component checks for messages before each of its deduction operations. This frequent checking in the hyper-resolution component is necessary as in some problems a single nucleus may be in use for a long time. If the checking is reduced to only when a new nucleus is chosen, then the hyper-resolution component does not take sufficient advantage of the clauses created in the other components.

Within each deduction component, each clause that is created is passed to a clause control module in the component. The clause control module implements the subsumption checks, removes subsumed input clauses from the local input set, adds the new clause to the local input set, and transmits both the new clause and subsumed clauses' identifiers to the controller. The <type> of a new clause message is `clause`, and the <data> field contains the new clause and the subsumed clauses' identifiers. When the controller receives a message of this type it forwards the new clause and the subsumed clauses' identifiers to other deduction components, also in a `clause` message. The <source> field enables the controller to selectively forward the <data>, e.g. clauses received from the hyper-resolution component are not forwarded to the GLD component. Upon receipt of `clause` messages, the deduction components remove the subsumed clauses from their input sets, and add in the new clause.

The `stop` message type is used when a deduction component either finds a refutation, or completes the search of its search space. If a deduction component finds a refutation it sends a `stop` message to the controller, with a <data> value of `success`. The successful component then proceeds to terminate. The controller forwards the `stop` message to the other deduction components. Upon receipt of these messages the deduction components proceed to terminate. If a deduction component completes the search of its search space without finding a refutation, it sends a `stop` message to the controller with a <data> value of `failure`. If the failing component has a complete deduction format then it proceeds to terminate, and the controller forwards the `stop` message to the other deduction components. If the failing component is incomplete, then the `stop` message is ignored by the controller as a refutation could still be found. The failing component does not terminate is this situation, as it could receive a clause from another deduction component hence opening up another piece of search space. Therefore incomplete deduction components, that have completed their search, wait for further messages from the controller.

The `io` message type allows the deduction components to output data at the controller's terminal. This is useful for tracking the progress of the HPDS. Whenever the controller receives an `io` message, the <data> is printed to its standard output.

When a deduction component proceeds to terminate, it determines the number of deduction operations it has performed and the amount of CPU time it has used. These figures are sent to the controller in a `statistics` message. After the controller has sent a `stop` message to each deduction component, it collects the `statistics` messages and outputs appropriate statistics. The final task of the controller after this is to remove any messages left in the tuple space.

Testing

To evaluate a parallel deduction system, it is important "to state UNAMBIGUOUSLY ... what one wants to measure" [Suttner,□1992]. For the HPDS, the testing has measured□:

- The amount of *processing* the HPDS performs to find refutations. Processing is measured in terms of CPU Units, where one CPU Unit is the amount of processing done on one processor in one unit of time.
- The amount of *time* the HPDS uses to find refutations. Time is measured in seconds.

The processing and time usages of the HPDS have been compared with those of five other deduction systems. They are□:

1. The GLD component running independently.
2. The UR component running independently.
3. The hyper-resolution component running independently.
4. A naive version of the HPDS, in which all the components are run time sliced on a single processor. The only inter-component communication in the naive system is to stop all components as soon as any one finds a refutation. The processing and time usages of the naive system are calculated from from those of the independent components.
5. An 'average system', whose processing and time usages are the averages of those of the independent components.

The HPDS and the independent components have been tested on 132 problems. Due to the large number of problems, a time limit of 500 seconds was imposed on all tests. In 49 of the problems none of these systems found a refutation within the time limit. The results for the remaining 83 problems have been analysed. For each of the five systems listed above, the ratios of its and the HPDS's processing and time usages have been determined for each problem. The ratio of the processing usages is called the *productivity*. The ratio of the times taken is called the *speed-up*. Two speed-up ratios are considered. The first ratio compares the HPDS with the other system running on a processor whose processing capability combines the three processors used by the HPDS. The second ratio compares the HPDS with the other system running on one of the processors used by the HPDS. The first ratio measures speed-up due to strategy shift, while the second measures speed-up due to parallelism and strategy shift. In all cases a value greater than 1 indicates that the HPDS is performing better than the other system. The results are summarised in the following table.

Measure	Compared With				
	GLD	UR	Hyper	Naive	Av.
HPDS & ~2nd	9	23	25	3	3
HPDS & 2nd	63	49	47	69	69
~HPDS & 2nd	9	7	1	11	11
Av. Prod'y	1.60	0.96	7.83	1.13	2.05
# Prod'y > 1	15	4	12	12	12
Av. SU(1)	0.98	0.48	4.80	0.64	1.24
# SU(1) > 1	10	3	5	8	9
Av. SU(2)	2.95	1.49	14.4	1.93	3.71
# SU(2) > 1	26	8	15	39	14

Legend

HPDS & ~2nd: The number of problems for which the HPDS found a refutation and the other system did not.
HPDS & 2nd: The number of problems for which both the HPDS and the other system found a refutation.
~HPDS & 2nd: The number of problems for which the other system found a refutation and the HPDS did not.
Av. Prod'y: The average productivity, measured over the problems for which both the HPDS and the other system found a refutation.
Prod'y > 1: The number of problems for which the productivity is greater than 1.
Av. SU(1): The average speed-up, measured over the problems for which both systems found a refutation, with the other system running on a combined processor.
SU(1) > 1: The number of problems for which the speed-up(1) is greater than 1.
Av. SU(2): The average speed-up, measured over the problems for which both systems found a refutation, with the other system running on a single processor.
SU(2) > 1: The number of problems for which the speed-up(2) is greater than 1.

Performance Comparison Table

The HPDS finds refutations for 72 of the 83 problems analysed. The GLD component finds the refutation in 40 of the 72 problems, the UR component in 28 problems, and the hyper-resolution component finds only 4 refutations. The unit clauses produced by the UR component are believed to be a major contributing factor to the strong performance of the GLD component in the HPDS. Conversely, the clauses passed to the UR component in the HPDS enable the UR component to find the refutation in six problems which it cannot solve independently. This is clear evidence of fertilisation from the GLD and hyper-resolution components. The hyper-resolution component appears to contribute the least to the HPDS. The role that the hyper-resolution component may be playing is to improve the performance of the UR component by supplying a large number of nuclei. If this is the case, and it is also true that the UR component's unit clauses are benefiting the GLD component, then the hyper-resolution component plays an integral role in the HPDS. A version of the HPDS, in which only the GLD and UR components are used, is being tested to determine the extent to which this is true.

The figures above show that the HPDS is more productive than the other systems. The less than 1 average productivity when compared to UR deduction, is offset by the 23 problems for which

UR deduction does not find a refutation. In terms of speed-up, the HPDS is faster than all the other systems when they are run on a single processor (speed-up(2)), as should be expected. If the comparison 'plays fair', and provides the same amount of processing power to both the HPDS and the other systems (speed-up(1)), then the HPDS is less dominant. In particular, the naive system performs better. A speed-up(1) value greater than 1 is roughly equivalent to super-linear speed-up, as it is commonly defined. Thus the HPDS displays super-linear speed-ups against each of the other systems, for some problems. Again, all the speed-up figures should be viewed in conjunction with the number of problems for which the other system does not find a refutation.

Conclusion

The HPDS is one of a large class of parallel deduction systems whose architecture is identified by the relative independence of the deduction components that run in parallel. The current version of HPDS is an early development in this class of systems, and there is clearly scope for further investigation. Many questions concerning appropriate combinations of components have yet to be addressed. The Prolog-D-Linda environment makes it possible to quickly and easily build and evaluate combinations of components. Experiments with different combinations of components are, to a large extent, unhampered by the difficulty of determining the compatibility of the individual deduction formats. This is in contrast to the difficulties experienced when combining multiple refinements of resolution into a single deduction system.

Besides experiments with different deduction component combinations, there is also scope to increase the level of cooperation between the component of the HPDS. In the current configuration the deduction components are in cooperative competition. However, the cooperation consists of simply distributing clauses without knowledge of whether the clauses will benefit the recipients. In an improved HPDS, the deduction components would cooperate reactively and proactively with each other. A deduction component with particular skills could be requested to try create a particular clause, by another component without those skills. The deduction components would also be more aware of their own needs, and would carefully filter clauses created in other components. The current HPDS has only one instance of this type of filtering, where the GLD component does not accept clauses from the hyper-resolution component. The performance of the HPDS was significantly worse before this filter was added. Closer cooperation would also produce more stable results. The performance of the current HPDS is affected by very small changes in the timing of clause distribution, brought on by varying processor loads.

Finally, the strong performance of the naive system, relative to the HPDS, suggests that heterogeneous competitive systems are worth investigating.

References

Ertel W. (1990), Random Competition: A Simple, but Efficient Method for Paralleizing Inference Systems, 342/27/90 A, Institut fur Informatik, Technische Universitat Munchen, Munich, Germany.

Loveland D.W. (1969a), A Simplified Format for the Model Elimination Theorem-Proving Procedure, In *Journal of the ACM* 16(3), ACM Press, New York, NY, 349-363.

Lusk E.L., Slaney J.K., and McCune W.W. (1991), ROO, In Kanal L.N., Suttner C. B. (Ed.), *Informal Proceedings of PPAI-91, International Workshop on Parallel Processing for Artificial Intelligence* (Sydney, Australia, 1991), International Joint Conferences on Artificial Intelligence, Inc., Sydney, Australia, 110-116.

Overbeek R., McCharen J., and Wos L. (1976), Complexity and Related Enhancements for Automated Theorem-Proving Programs, In *Computers and Mathematics with Applications* 2, Pergamon Press, England, 1-16.

Robinson J.A. (1965b), Automatic Deduction with Hyper-resolution, In *International Journal of Computer Mathematics* 1, Gordon and Breach, London, England, 227-234.

Shostak R.E. (1976), Refutation Graphs, In *Artificial Intelligence* 7, Elsevier Science, Amsterdam, The Netherlands, 51-64.

Sutcliffe G. (1991), A Parallel Linear and UR-Derivation System, In Kanal L.N., Suttner C. B. (Ed.), *Informal Proceedings of PPAI-91, International Workshop on Parallel Processing for Artificial Intelligence* (Sydney, Australia, 1991), International Joint Conferences on Artificial Intelligence, Inc., Sydney, Australia, 211-215.

Sutcliffe G. (1992), The Semantically Guided Linear Deduction System, In Kapur, D. (Ed.), *Proceedings of the 11th International Conference on Automated Deduction* (Saratoga Springs, NY, 1992), Springer-Verlag, New York, NY,

Sutcliffe G., and Pinakis J. (1991), Prolog-D-Linda : An Embedding of Linda in SICStus Prolog, 91/7, Department of Computer Science, The University of Western Australia, Perth, Australia.

Suttner C.B. (1992), Personal correspondence about evaluating parallel deduction systems.

Wos L., Carson D., and Robinson G.A. (1964), The Unit Preference Strategy in Theorem Proving, In *Proceedings of the AFIPS 1964 Fall Joint Computer Conference* (San Francisco, CA, 1964), Spartan Books, Baltimore, MD, 615-621.