A Parallel Resolution Procedure Based on Connection Graph

P. Daniel Cheng
Advanced Information Services Inc.
1512 Candletree Dr.
Peoria, IL 61614

J. Y. Juang
Dept. of Electrical Engineering & Computer Science
Northwestern University
Evanston, IL 60201

ABSTRACT

In this paper, we present a new approach towards a parallel resolution procedure which explores another dimension of parallelism in addition to the AND/OR formulation and special hardware constructs. The approach organizes the input clauses of a problem domain into a connection graph. The connection graph is then partitioned and each partition is worked on by a different processor of a multiprocessor system. These processors execute the resolution procedure independently on its partition, and exchange intermediate results via clause migrations.

Preliminary test results and qualitative assessments of this procedure are also given.

1. Introduction

Resolution procedure has been the basis of automatic theorem-proving and logic inference since its first introduction in 1965 [1]. However, its execution on today's computers is too slow to be effective, primarily due to the long resolution cycle time and exponential nature. Although exponential explosion remains unavoidable, several parallel schemes have been proposed to improve the speed performance of the resolution process. Among them, the most current topic is the approach of AND/OR parallelism. However, because of the impedance of shared variables between AND branches and the small number of OR branches found in most existing programs [2-4], concurrency from AND/OR parallelism approach is very limited in practice.

In this paper, we propose a new approach towards a parallel resolution procedure which in essence explores another dimension of parallelism in addition to the AND/OR formulation and special hardware constructs. The approach organizes the input clauses of a problem formulation into a connection graph [5]. The connection graph is then partitioned and loaded into multiple processors. These processors execute the resolution procedure independently on its partition, and exchange intermediate results via clause migrations. The construction of connection graph is described in Section 2. A resolution procedure based on this graph is also briefed in this section. In Section 3, we present a parallel model for executing the procedure on multiprocessor systems. The parallel procedure is evaluated in Section 4, and conclusions are drawn in Section 5.

2. Resolution Based on Connection Graph

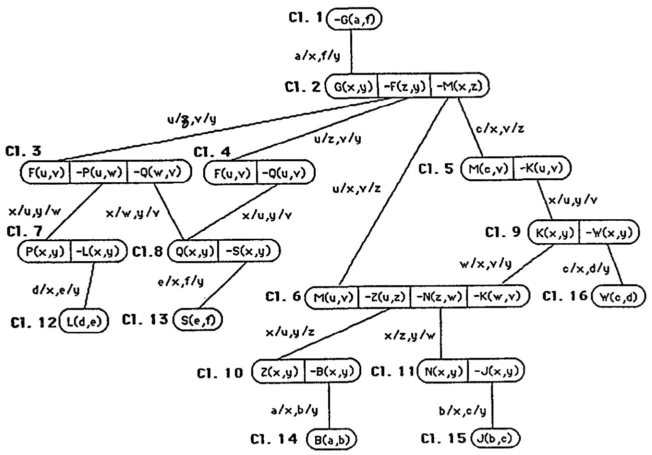
2.1 Graph Representation

A graph structure of an input clause set is constructed as follows: each literal of clauses in the input clause set is represented as a node in this graph, and the nodes representing literals of a clause are grouped together. Unification is then conducted to match every pair of literals which have the same predicate symbol and are complementary in sign. If the unification attempt between two literals succeeded, these two unifiable literals are marked by a link and the resulting MGU (the most general unifier) is used to label this link. Given the clause set of Figure 1.a, the corresponding graph structure is shown in Figure 1.b.

The graph representation offers several merits over those represented in plain clause set. Among them, the most notable one is the clause matching process in which clauses unifiable with a key clause are to be identified in each resolution step. Using the plain clause set representation, a set-wide search is needed every time a key clause is presented. Although some efficient data structures can be imposed (e.g., the FPA [6] lists) to restrict the search on relevant clauses only, unification still has to be performed on each candidate clause and is subject to failure. Furthermore, the complexity each time is proportional to the number of clauses at that state.

Cl. 1. -G(a,f)	Cl. 9. K(x,y) -W(x,y)
Cl. 2. G(x,y) -F(z,y) -M(x,z)	Cl. 10. Z(x,y) -B(x,y)
Cl. 3. F(u,v) -P(u,w) -Q(w,v)	Cl. 11. N(x,y) -J(x,y)
Cl. 4. F(u,v) -Q(u,v)	Cl. 12. L(d,e)
Cl. 5. M(c,v) -K(u,v)	Cl. 13. S(e,f)
Cl. 6. M(u,v) -Z(u,z) -N(z,w) -K(w,v)	Cl. 14. B(a,b)
Cl. 7. P(x,y) -L(x,y)	Cl. 15. J(b,c)
Cl. 8. Q(x,y) -S(x,y)	Cl. 16. W(c,d)

(a) The input clause set



(b) Graph representation of input clause set

Figure 1. An example problem.

Since the number of clauses grows rapidly during the resolution process, this turns out very inefficient. This problem, however, can be eliminated using graph representation in which unifiable clauses are dynamically maintained and the associated MGUs are immediately available.

For each new clause generated by resolving upon one of the link, clauses possibly unifiable with the resolvent can be easily identified through the links of its parent clauses. No extensive matching is needed and more importantly, most of the new MGUs of the resolvent's links can be simply obtained through composition of substitutions.

2.2 Resolution on Connection Graph

After the connection graph is constructed, the resolution procedure then repeatedly selects a link, resolves upon this link, generates the associated resolvent, and finally inserts this resolvent into the connection graph. This process repeats until a null resolvent is generated or no more resolution is possible. This process is outlined in Figure 2.

```
A connection graph is solved if it contains the empty clause
To solve a connection graph which does not contain the empty clause
    if there is a clause containing an unlinked literal
        delete this clause together with its associated links
    otherwise
        select a link
        delete the link and generate the resolvent
        if the resolvent is a tautology
            delete the resolvent
        otherwise
            add the resolvent together with its new links to the graph
    solve the resulting connection graph
```

Figure 2. The sequential resolution procedure.

Each resolvent generated inherits the unifiable links from its two parent clauses, and the new MGUs of these links are obtained by the composition of the old MGU and the MGU used in the resolution. Substitution compatibility is checked in the mean time and incompatible links are not inherited. After the resolvent and its links are generated, the link previously used to conduct the resolution is removed from the two parent clauses.

If the resolvent is not an empty clause, it is checked for deletion due to *tautology* or *pure literals*. Because tautologies do not positively contribute to the solution of problems, they can be deleted from a set of clauses without affecting the inconsistency. A literal in the resolvent becomes pure when it fails to inherit any link from the parent clauses. A clause containing a pure literal can not contribute to a refutation because the unlinked literal can never be resolved upon [1]. Either parent clauses can also become pure after the removal of the resolution link. These clauses are subsequently deleted from the connection graph.

Deletion of clauses containing pure literals is an important feature of the connection graph proof procedure. In addition to the clause itself, all links connected to its literals must also be deleted from the graph. Deletion of such links, however, may cause literals in other clauses to become unlinked. Thus deletion of clauses can create a chain reaction in which a succession of clauses is deleted from the graph. Deletion of clauses simplifies the connection graph, reduces the search space, and makes it easier to find a solution.

3. Parallel Resolution on Connection Graph

We take the algorithmic approach of the connection-graph procedure described above in formulating a parallel resolution procedure. For the resulting procedure, we impose no special architecture requirements; therefore, any speed advantages obtained from hardware enhancements can also be incorporated.

3.1 The Parallel Approach

In conventional parallel resolution procedures, clauses are stored at shared memory, and all the processors access the same store to obtain a clause pair (see Figure 3.a). This approach incurs serious memory conflicts, and results in a very long resolution cycle. To reduce resolution cycle time, clauses can be partitioned into smaller subsets. Each is stored in the local memory of a processor, and resolved by the processor in parallel with others (see Figure 3.b). A clause set can be partitioned in such a way that each subset forms a conceptual cluster. Therefore, each processor can concentrate on a concept and keep busy all the time. Nevertheless, a subset so obtained may not always contain sufficient clauses for a successful proof. It has to request necessary clauses from others from time to time as resolution proceeds. The migration of clauses adjusts the partition dynamically so that a proof can be found by one of the participating processors. Thus, clause migration is essentially a robust scheme that explores conceptual clusters automatically. Via clause migrations, a processor in the above procedure conducts resolution virtually on the whole clause set, though it is in fact working only on a small subset of clauses. Thus, this parallel resolution procedure can be seen as a form of *virtual resolution mechanism*. This allows processors to work on the same

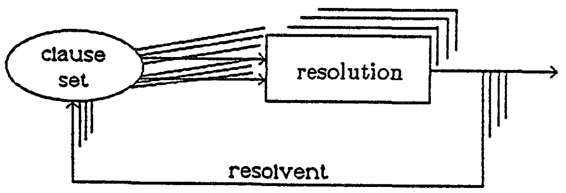


Figure 3.a A conventional parallel resolution procedure with clauses stored in shared memory

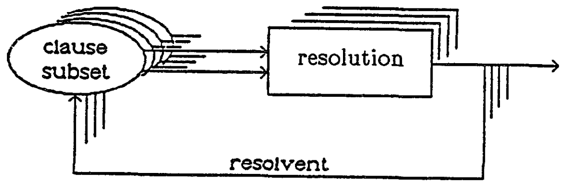


Figure 3.b Proposed parallel resolution procedure with partitioned clause subsets

search path, which is impossible in the AND/OR tree search procedure. Furthermore, neither shared variable nor synchronization is necessary.

3.2 Initial Graph Partition

In response to the problem decomposition in parallel processing, the first task of this parallel procedure is the decomposition of the initial connection graph. The general rule of problem decomposition in parallel processing is to allow as much parallelism and as less interprocessor communication as possible. However, because of the non-deterministic nature of the resolution process, fully loading each processor with a sufficient workable subtask is not necessarily most productive. Although the communication overhead is our concern, least interprocessor communication alone can't be efficient either. In the context of theorem-proving or logic inference, resolution process is usually guided by a heuristic in order to get a prompt proof. Problem decomposition should also follow this discipline such that each subtask can work effectively and cooperatively, not just fully utilize the processor resources.

In this version of the parallel model, we provide a preliminary scheme of problem decomposition through which the initial connection graph is decomposed into distinct partitions. First of all, we assign each link a preference measure whose value is determined based on the resolution strategy or heuristics in use. For example, if unit preference strategy is used, the preference measure of a link can be directly set to the no. of residual literals of that link. If set-of-support (SOS) strategy is used, preference measures of links can be placed at three different levels depending on whether both of the linked clauses are supported, only one of them is supported, or neither is supported. These levels can differ by an order of magnitude with a secondary strategy, e.g., unit preference, ordering the links within a level. Notice that these preference measures can be used in selecting links during the resolution process as well.

After the preference measures are established, an inclusion process is invoked to group clauses starting from some seed clauses. Unit clauses or clauses having support can be used as the seeds and potentially, one partition will grow from each of the seeds. The inclusion process will run on each partition in turn and allocate one clause to that partition at a time. (Multiple allocation may be desirable in some cases.) During the inclusion process, clauses adjacent to that partition are identified first. The clause with the best preference measure and not allocated is then included to that partition. (For the case of multiple inclusion, clauses having the same preference measure can all be allocated.) Contention of clauses, i.e., clauses having the best preference measure but were allocated to other partitions, is also marked and used later to determine the final partition pattern.

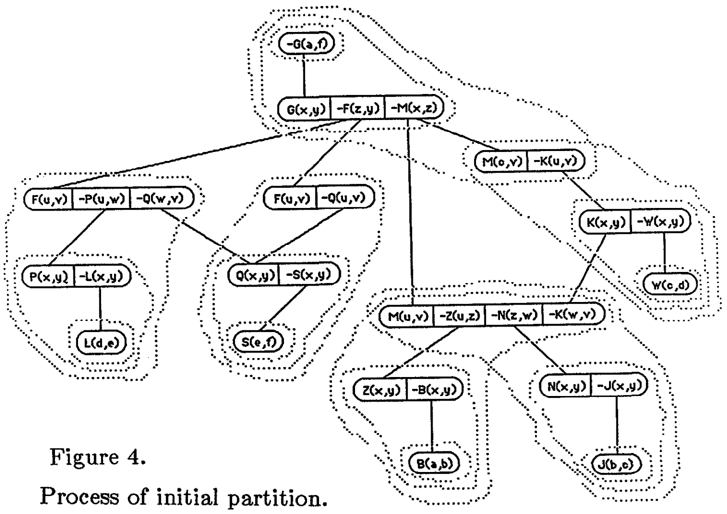


Figure 4.

Process of initial partition.

The basic philosophy behind this inclusion process is to avoid the situation that a clause is allocated to a partition and has no links with any clause of that partition. A clause under this situation is called an *isolated clause* in this paper. Therefore, only clauses linked with that partition are considered at each step of the inclusion process. The inclusion process terminates when all the input clauses are allocated. Executing this process on the connection graph of previous example is illustrated in Figure 4, where six unit clauses are used as the seeds.

Each partition thereafter formed can ideally be used as a subtask for the parallel resolution. However, we further analyze the overlapping of clauses between partitions to determine the optimal partition pattern. A basic criterion is that if two partitions have a moderate degree of overlapping and each is small in terms of the no. of clauses, we merge these two partitions into one as illustrated by the merge of two partitions in Figure 4, where finally four final partitions are formed, see Figure 5. Partitions with extensive overlap are merged to reduced the communication overhead. Partitions with small no. of clauses are merged in order to maintain a feasible no. of clauses in each partition and to avoid processors running out of clauses.

3.3 The Parallel Resolution Procedure

After the initial connection graph is decomposed, each partition is loaded into a different PE of a multiprocessor system for execution. Each of these PEs will perform the conventional connection graph proof procedure on its local partition, together with from time to time interprocessor communications.

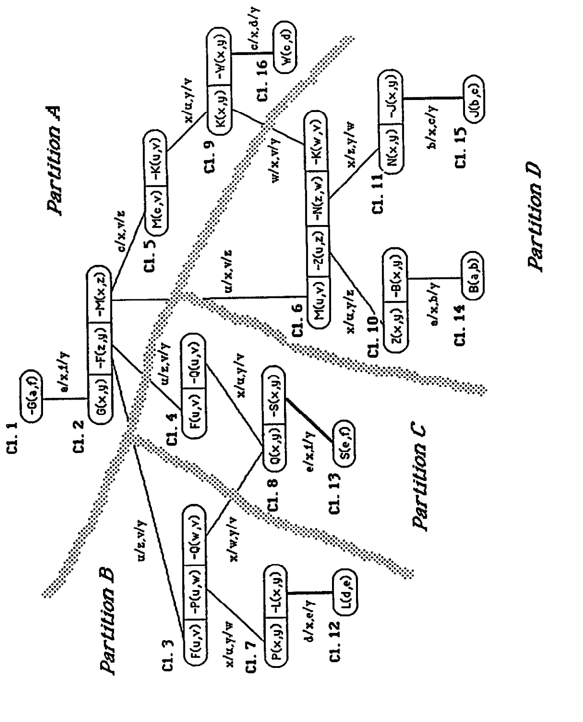


Figure 5. Initial partition on example clause set.

Typically, an interprocessor communication need arises when the linkage structures between partitions change. For example, when a resolvent is generated, its new external links are established through the interprocessor communication. If a clause is deleted due to pure literals or subsumption, all of its external links are also broken through the interprocessor communication. For the needs of these communication occasions, a protocol set is designed to handled these works [7]. This protocol is asynchronous type, and is running as a child process of the resolution process in the current design. If a hardware module can be built around it, the communication delay can be significantly reduced.

During the resolution process, each PE will repeatedly select a link (based on the preference measures), generate the associated resolvent, and update the graph structure. This process repeats until an empty clause is generated. The news of empty clause is immediately broadcast to all other PEs to stop the whole process. This broadcasting is done through the communication protocol also. If there exists no empty clause for the problem, manual interruption is needed.

3.4 Clause Migration

In each cycle of the resolution process, a link is selected from those belonged (completely or partially) to the local partition. If the link selected is an external link, this indicates that the local resolution has run to the point where a remote clause can contribute to the local resolution. In response to this, we provide the clause migration mechanism through which clauses are transferred between partitions. Through this mechanism, we survive the problem of completeness resulted from the decomposition of the input clauses. Furthermore, if the remote clause is an intermediate result of other partition, we get the intended speedup by having someone else doing that derivation.

The generation of isolated clauses is another occasion for clause migration where all the internal links of a clause are resolved away by local resolution. Since isolated clauses are no longer useful in the local partition, it is desirable to transfer them to other partitions. In determining the destination of an isolated clause, we can migrate the isolated clause to the partition which has the largest no. of links with it, or to the partition which has the maximal preference value on the link. The former potentially minimizes the communication overhead afterward while the latter could be more effective to the whole resolution procedure.

We summarize our introduction of this parallel procedure in the following algorithm where highlighted steps are intended for comparison with the sequential procedure of Figure 2.

The rate of clause migrations is considered an overhead in this parallel procedure and it can be minimized through a proper decomposition in the initial partition stage. The procedure we devise for initial partition is found satisfactory. Also worth mention here is the lock procedure embedded in the clause migration mechanism. That is, before a clause can be migrated, all clauses linked with it are locked from being resolved. This avoids losing track of the link structures while clauses are migrating. It also prevents the situation that two clauses are migrating to each other. Although better schemes can be devised to get around this restriction, it is this method used in current design.

```
Decompose the initial connection graph into partitions
A global connection graph is solved if one of its partitions
 contains the empty clause
To solve a global connection graph which does not contain
 the empty clause
    (* solve each partition concurrently *)
    PARA
        if there is a clause containing an unlinked literal
            delete this clause together with its associated links
             from the global graph
        otherwise
            select a link
            migrate the remote clause if an external
             link were selected
            delete the link and generate the resolvent
            if the resolvent is a tautology
                delete the resolvent
            otherwise
                add the resolvent together with its new links
                 to the global graph
        migrate the isolated clauses
        solve the resulting global connection graph
```

Figure 6. The parallel resolution procedure.

4. Performance Evaluation

A preliminary performance assessment of this parallel procedure is conducted based on a series of program verification problems suggested by McCharen et al [8]. The execution of the parallel procedure is emulated by a simplified prototype which creates one logic process to simulate a physical processor. In this prototype, clause migration takes place only when a clause is isolated in a partition. The solution time, in terms of resolution step, is used as the primary measurement. Each problem is solved several times by slightly varying the number of partitions in order to observe the performance fluctuation under different partition numbers. The solutions with single partition are used to resemble what would have been obtained from the sequential procedure. The test results are summarized in Table 1.
because of the restriction of migration upon isolation only in current implementation, a clause needs to exhaust all of its internal links to becomes isolated and available to other partitions. This may generate a vast amount of clauses in the local PE and delay its timely effect on other PEs.

5. Concluding Remarks

We have described a new approach to the parallelism of resolution procedure for theorem-proving and logic inference. The approach explores another dimension of parallelism in addition to the pipelined architecture constructs and the AND/OR parallelism. The control over the individual clause level also provides us the flexibility in incorporating existing resolution strategies developed from the theoretic study of theorem-proving. The formulation of this parallel model does not either impose any special hardware requirement, and can thus be easily realized on any multi-computer system or local computer network. Observing that this parallelism is only limited by the number of PEs and the communication support, an ultimate speedup can be achieved when the resolution process is guided by an elaborate strategy.

Secondly, we want to address the innovation idea of clause migration. This clause migration capability supports the effectiveness of resolution strategies, and provides an illusion of virtual resolution even though the clauses are distributed over different sites. That is, as soon as a clause becomes material to the resolution process of one partition, this clause can be made available to that partition immediately without any concern of its residency.

Finally, from this investigation we also identified another advantage of the connection graph representation which we do benefit in the formulation of this parallel procedure. The link structure of the connection graph facilitates our work of clause partition by providing us information about clause interrelation. With these clause interrelations established beforehand, every effort can be made to group relevant clauses into the same partition. Each link itself is also an indicator of how heavy a clause is relative to other clause. Thus, communication overhead can be reduced by simply minimizing the number of links between partitions. With only these links maintained in each partition, conversations between partitions are easily conducted along these links without knowing each other's whole clause set whatsoever.

Also implied in our presentations are several enhancements to the parallel procedure, like programming each PE to use a different resolution strategy, relaxation of the lock restriction, dynamic partition split on heavy-loaded PEs, and finally the complete realization of this parallel procedure on a real multiprocessor system. Those are the major topics of our further research on this parallel approach.

REFERENCES

[1] J. A. Robinson, "A Machine Oriented Logic Based on the Resolution Principle," *JACM*, vol. 12, pp. 23-41, Jan. 1965.

[2] S. J. Stolfo and D. Miranker, "DADO: A Parallel Processor for Expert Systems," in *Proc. 1984 Int'l Conf. Parallel Processing*, pp. 74-82, Aug. 1984.

[3] J. S. Conery and D. F. Kibler, "AND Parallelism and Nondeterminism in Logic Programs," *New Generation Computing*, vol. 3, pp. 43-70, 1985.

[4] K. Murakami, T. Kakuta, and R. Onai, "Architecture and Hardware System: Parallel Inference Machine," in *Proc. Int'l Conf. Fifth-Generation Computer Systems*, pp. 18-36, Tokyo, 1984.

[5] R. Kowalski, *Logic for Problem Solving,* North-Holland, New York, 1979.

[6] E. Lusk and R. Overbeek, "Data structures and control architecture for the implementation of theorem-proving programs," in *Proc. 5th Conf. Automated Deduction*, pp. 232-249, 1980.

[7] P. Daniel Cheng, *A Parallel Theorem Prover Based on Connection Graph,* Master's Thesis, Nothwestern University, Evanston, Illinois, Dec. 1986.

[8] J. D. McCharen, R. A. Overbeek, and L. A. Wos, "Problems and Experiments for and with Automated Theorem-Proving Programs," *IEEE Trans. Comput.*, vol. C-25, pp. 773-781, Aug. 1976.

Initial No. of Clauses	SOS	Resolution Steps Taken						Best Speedup*	Normalized Speedup
		AURA	No. of Partitions						
			1	2	3	4	5		
14	all	108	42	29	39	33	-	3.7α	1.8
32	all	134	90	78	50	55	53	2.7α	0.9
27	$\neg$Conc	750	387	-	317	264	257	$>2.9\alpha$	0.58
29	$\neg$Conc	342	213	-	250	162	194	2.1α	0.52
24	$\neg$Conc	130	78	68	50	61	143	2.6α	0.87
24	$\neg$Conc	133	69	59	51	44	-	3.0α	0.75
27	$\neg$Conc	483	303	-	412	211	192	$>2.5\alpha$	0.5
21	$\neg$Conc	191	96	83	67	49	75	3.9α	0.98

Table 1. Results of testing the proposed model on a set of program-verification problems in non-Horn clauses.

<Note> The speedup of the proposed model over AURA is adjusted by a factor ($\alpha > 1$) to account the following two facts: (1) Hyper-resolution is used in AURA, which may resolve more than one pair of literals in each step; (2) Resolution cycle of proposed method is shorter than that of AURA since no string matching is necessary.