Parallelization of Temporal Logic Verification by Dividing State Set

Shinji Kono, Masayuki Ikemura

e-mail:kono@ie.u-ryukyu.ac.jp
Information Engineering, University of the Ryukyus
Nishihara-cyo 1, Okinawa, 903-01, Japan

January 15, 2000

Interval Temporal Logic is useful in software / hardware verification. Deterministic tableau expansion is used but it requires P-space complete computational complexity. Especially state space explosion is critical. In order to make large specification verifiable, each temporal logic state is stored as Binary Subterm Tree and divided into PC cluster. By division, state space consists of small Binary Subterm Tree, which can be stored efficiently and can be accessed in fast way. The tableau expansions are performed in parallel. We observed super linear acceleration in some case. The system Plite is written Prolog and UDP stream library.

1 Interval Temporal Logic: ITL

Here we use Interval Temporal Logic (ITL), more specifically, Local ITL, in which value of variables are depend only on clock period [4]. This logic has discrete time and time interval. It also has the end of the time, but it may contains infinite clock period. Unlike LTTL (Linear Time Temporal Logic) , ITL cannot express fairness such as "It happens infinitly many often' ', but verification procedures is easier than LTTL.

Other than classical logic operator (and, or not), there are three temporal logic operator in ITL.

- $@P$: At the next interval, P happens. If there is not next interval, the formula is F. (next)
- $P\&Q$: An interval is divided and P is T in the former interval and Q is T in the later interval. (chop)
- *empty* : T if the length of the interval is 0.

We can express sometime-$\Diamond$ and always-$\Box$ using chop operator likes this;

$$\Diamond P \equiv T\&P \text{sometime}$$
$$\Box P \equiv \neg(T\&\neg P) \text{always}$$

empty can be represented by $@T$, but we treated *empty* as a basic operator. In this paper, we assume ITL formula is in chop normal form, that is, its $\Diamond, Box$ operators are all translated by chop operators.

1.1 What can be verified in ITL verification?

For an ITL formula, using Deterministic Tableau Expansion, a deterministic finite state automaton is generated. It accept event sequences which makes the formula T. This is an automatic program generation from ITL specification. (But in order to execute it, some backtrack mechanism is necessary.)

It is easy to extends ITL description to represent finite state machine (adding by *-operator) or to define tableau expansion for finite state machine. So we can do model checking directly by verifying a formula like this;

$$\text{implementation} \rightarrow \text{specification}.$$

Counter examples of the formula means implementation error.

Since this is an automatic theorem prover, we can check validity or satisfiability of the formula.

2 Deterministic Tableau Expansion

Any ITL formula can be divided into two parts; current clock part and next clock part. Using *empty* and @-operator, this can be represented in a disjunctive normal form;

$$P \Leftrightarrow (empty \wedge P_e) \vee \bigvee_i P_i \wedge @X_i.$$

Here P_e, P_i contain no temporal logic operators. (But X_i is.) We call this $@ - normal\ form$.

X_i is an ITL formula which represents a state. The state expanded into $@ - normal\ form$ recursively. In local ITL, X_i is only finitely generated, and expansions are terminated eventually. We have finite state automaton which states are X_i and transitions are P_i. Since X_i represents a state, X_i is called state formula. Please note that P_i nor P_e is not a part of the state. These are transition conditions. These conditions are not important during tableau expansion.

It is also possible to make P_i exclusive each other in a state. This gives deterministic automaton and we call it Deterministic Tableau Expansion. Or we call it ITL tableau for short. In this case, we don't have to calculate rather expensive negation of disjunction n normal form. But the tableau method actually contains subset construction method which is a determinization of non-deterministic automaton. It may costs exponential of number of original state. But in our experience, subset construction is cheaper than negation of disjunction normal form.

An expansion rule is defined for each ITL operator. Many of them are trivial. For example, in case of $P\&Q$, we assume P is already expanded;

$$\begin{aligned} P &\Leftrightarrow (empty \wedge P_e) \vee \bigvee_i P_i \wedge @X_i \\ P\&\ Q &\Leftrightarrow (empty \wedge P_e \wedge Q) \vee \bigvee_i P_i \wedge @(X_i \& Q)(1) \end{aligned}$$

Here se use $(@P)\&Q \equiv @(P\&Q)$ also.

To make P_i deterministic, values of the variables in P_i is fixed. Using backtrack all possible combinations of value of variables in P_i are tried. P_i becomes set of signed propositional variables in P_i. $+P$ means P is T, $-P$ means P is F.

This means number of the state transition is 2^n, where n is number of the variables in P_i. So this method does not work well if number of the variables is too large. But it is robust against complexity (nesting of temporal logic operator) or length of ITL formula.

2.1 Binary Subterm Diagram

During expansion, X_i is generated as a state representation formula. It may very complex but using Binary Subterm Diagram we can make it simple for many cases. This is an extension of BDD[1].

Binary Subterm Diagram is binary tree which condition is a sub term of original formula. Here use use a notation;

$$?(Condition, Truepart, Falsepart).$$

This means if $Condition$ holds it means sub tree $Truepart$ otherwise it mans sub tree $Falsepart$.

For example, $\Diamond\Box p$ is translated into chop standard form; $T\&\neg(T\&\neg p)$. The formula is translated into Binary Subterm Diagram in a bottom up way. Each subterm have to be named. Because it is used as a condition part of subterm diagram.

$$\begin{aligned} T\&\neg(T\&\neg p) & \\ T\&\neg(T\&?(p,F,T)) & \quad \neg p =?(p,F,T) \\ T\&\neg(s_1) & \quad s_1 = T\&?(p,F,T) \\ T\&?(s_1,F,T) & \quad \neg(s_1) =?(s_1,F,T) \end{aligned}$$

The subterm diagram is expanded by the tableau procedure. In this case, resulted formula is $\neg(T\&\neg p) \vee (T\&\neg(T\&\neg p))$. This is rather complex formula, but its subterm diagram form is very simple; $s_3 =?(s_2, T, ?(s_1, F, T))$. (fig .1)

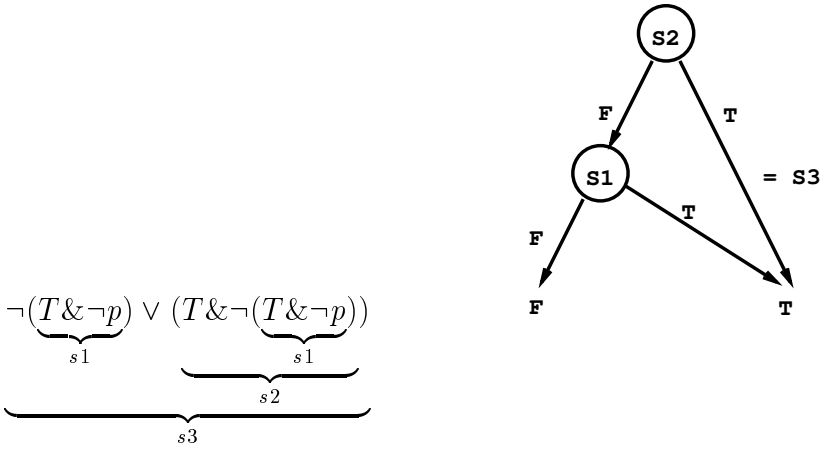

Figure 1: Binary Subterm Tree

Possible suterms in a formula are finite, but unlike classical tableau, ITL tableau may generates new subterms. It is easy to show generated subterms are finite in ITL tableau. Proof is an induction on nesting of modal logic operators. First, if it contains no temporal logic operator, its subterm

is finite. If there is a temporal logic operator, for example, in the chop expansion rule (1), $P\&Q$ generates new subterm; $(X_i\&Q)$. Each X_i is generated by the expansion of P and these are finite by the induction assumption, so $(X_i\&Q)$ are. If we have more complex temporal logic operators such as projection, the proof becomes complex, but basically the same proofs [**?**].

Now state space in ITL tableau is represented by named subterms and binary subterm diagram. If named subterms are correctly ordered, using maximum sharing, we have a unique representation of a formula. But even if naming is duplicated correctness of ITL tableau procedure itself is unaffected.

3 Parallelization of verification procedure

Parallelization is based on dividing state space. Each processor process the tableau procedure in a part of state space. This is a kind of data parallel algorithm.

Division of state space uses a hash function on binary subterm diagram. Named subterm have to be consistent in all the processors, otherwise uniqueness of binary subterm diagram is lost. This means naming procedure have to be done by a processor. The procedure assigns unique ID for each newly created subterm.

Here we use simple hash function as a starting point. `number(C)` is the unique ID for subterm C.

```
hash(true)  = 1
hash(false) = 2
hash(?(C,T,F))
      = number(C)+hash(T)*3-hash(F)
```

This hash function is not symmetric on negation and it keeps hash value of false part's leaves.

A generated state formula is translated into binary subterm diagram. Then its hash value is calculated. It is send to each processor based on the hash value. (Fig. 2)。 The generated formula is also store into own state formula database to prevent retransmission of the same formula.

When a state formula is received by a processor, it checks if it is already in own state formula database. If not it is added to waiting tableau process queue.

During subterm diagram translation, new subterm may be generated. The term is assigned a unique ID by a processor; subterm server. Subterm server also keeps the number of unprocessed state formula. If it is 0, the verification is done.

The number of transmission of state formula is generated state formula minus own state formula. Asynchronous stream communication is used to achieve throughput.

A communication with subterm server is a transaction. Synchrnous remote procedure calls are used for the communication, which are assignment of unique ID for subterm and detection of termination.

4 Parallel verification algorithm

Here is a pseudo C code of the parallel verification of ITL.

```
if ( self == main ) {
  Translate given ITL formula into chop normal form;
  Translate the formula a Binary Subterm Diagram ;
  Put it into Queue ;
  Mark self active mark; ⇒ RPC
}
foreach (Processor) {
        while (there are active processors ) {
```

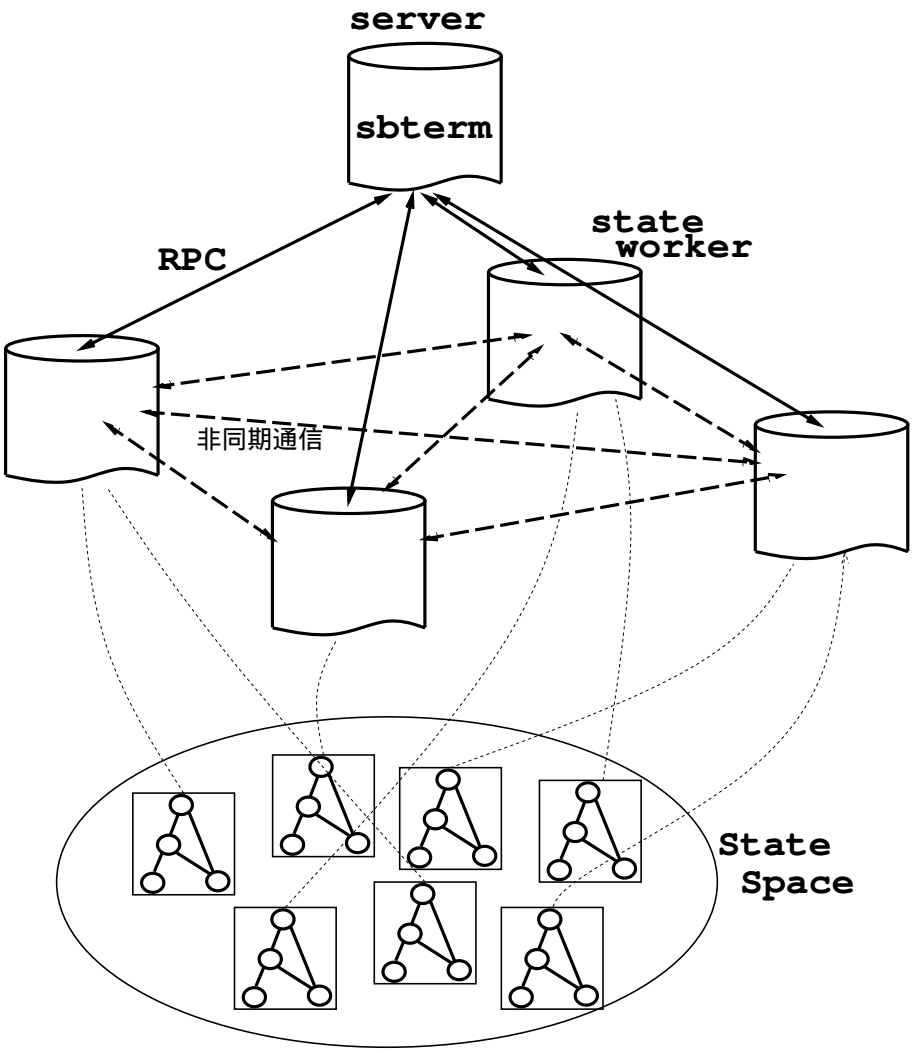

Figure 2: Parallel Verification Architecture

```
            if (Queue is not empty) {
              pick a Binary Subterm Diagram form the Queue;
              Tableau expansion(Binary Subterm Diagram);
            }
            mark self active ; ⇒ RPC
            check data from other processors; ⇐ State Stream
            check own state database;
            if (is new state formula) {
              Put it into Queue;
              PUt it into state database;
              mark self active ; ⇒ RPC
            }
        }
}
```

Definition of Tableau expansion is this;

```
Tableau expansion(Binary Subterm Diagram) {
  if ( a node is a subterm ) {
    check own subterm database;
    if ( no subterm in the database) {
      ( query on subterm database; ⇒ RPC
    }
  }
  if ( a node is a chop operator ) {
    do expansion in a recursive way;
    return resulted state formula;
  } else if ( a node is a @-operator) {
    remove @ and return it;
  } else if (a node is a variable) {
    chose true or false for the variable;
    (backtrack is used to achieve all truth value combinations;
  } else {
    it must be true/false;
  }
  Translate resulted state space subterm diagram;
  if (there is not subterm in own subterm database) {
    query on subterm server; ⇒ RPC
  }
  check own state database;
  if (is new state formula) {
    compute hash value for the state formula;
    if (is value for other processor) {
      send the state formula; ⇒ state stream
    } else
      put it Queue;
    store it into own state database;
  }
  perform backtrack on all possible truth value combinations;
}
```

4.1 Linda Communication Library

Linda is a tuple based communication library. Sicstus Prolog supports linda library written in Prolog. All Linda communication is held at a Linda server. It has simple tuple passing API.

`out(+Tuple)`	Places the tuple Tuple in Linda's tuple-space.
`in(+Tuple)`	Removes the tuple Tuple from Linda's tuple-space if it is there. If not, the pr
`in_noblock(?Tuple)`	Removes the tuple Tuple from Linda's tuple-space if it is there. If not, the pr

Figure 3: Linda Library

It is not suitable for large scale parallel processing since it has star network configuration. But it is easy to use. This is based on TCP/IP communication, library is quite reliable.

4.2 Datagram Communication Library

This system requires many-to-many communication in state formula trasmission. If we use channel type communication such as Unix stream socket, it requires N-1 communication channel for N processors. For large N, it eats kernel internal space. It also requires long loops to check channels. It also requires complex program for channel management.

We developed Datagram based in order to overcome this. It requires no channel initialization. It use only two sockets; one for state space communication and the other one for subterm server.

The library is written C and Prolog. It has Prolog Stream API, that is, we can handle Datagram packet using stream API. A packet is sent when the stream is flushed. It looks like stream, but actual communication is done by Datagram packet. Since we use Prolog Stream API, we need no special message trasmission syntax. But we have several addressing management API (Fig. 4)。

module setup	`?-use_module(library(datagram)).`
Stream Open	`?-datagram(+IP_host:IP_port,-Stream).`
Address registration	`?-datagram_address_registration (+Stream,+IP_host:IP_port,-Id).`
Select destination	`?-datagram_destination(+Stream,+Id).`
Packet check	`?-datagram_ready(+Stream,+Timeout,Ready).`

Figure 4: Datagram Library API

There is no reliability in Unix Datagram, but this library provides reliable communication. There are no packet size limit on the API, and Blocking / Deblocking is performed in the library. So Network's MTU (Maximum Transfer Unit) is not important at API level. The order of packet ais also preserved. In other words, this is a user level many-to-many TCP stream.

There is no connection either, but we have Address Database one for each Datagram Stream. We need one Database for each stream because Address Database contains buffers for reliable communication.

This library is very good with Sicstus Prolog's fast read/write library. In the fast read/write library, Prolog term is represented in an efficient way.

Transmission of state stream is performed asynchronously in some idle timing between tableau expansion. The library is tuned for throughput.

Accessing the subterm server is done by Remote Procedure Library, which is constructed on top of Datagram Library. So two datagram stream is necessary for each processor. We expect accessing subterm server is fewer than state formula communication, because subterm database is cached in each processor. So one subterm server is enough for this algorithm.

5 Experiments

This system is still under construction, but simple examples can be executed. Here we show three processors version for 5 dining philosophers specification in Datagram Library (Table 5). Speed up ratio of Linda communication for 3 dining philosophers is shown in Table 6.

We use MMX Pentinum 200MHz, 64MB memory, SICStus Prolog 3 #6.。 SICStus Prolog has no native compiler for Pentinum, WAM code interpreter is used. Databases are implemented by Prolog assert.

In this experiment, CPU time contains no idle time of data transmission. Actual computation time is more slower than this.

States are newly generated state in the CPU, which contains output cache of state formula. State tranmissions corresponds number of tableau expansions. Export/import means send/receive of state formulas. The difference of subterm number or state tranmissions are result of small difference of internal data structure.

In case of parallel processing, takes $3.28 = 7496.414/(725.896+843.71+713.055)$ speed up which is greater than number of CPU. This shows a kind of Super Linear Effect. But, in theory, there is no such effect. Actually sum of state transitions are equivalent. But in practice, data manipulation (modification or search) in small database is faster than large database. We think this effect comes from such kind of reason. We don't know if this Super Linear Effect still appears in other case yet.

In this example, number of state transition is much larger than number of state. This is a common situation, since state transition is corresponds to exponential of number of variables, but state is a combination of subterms. The actual ratio is about 6 in each processor. This fact justifies trasmission of state formula is lighter than computation of tableau expansion.

In case of empty interval, resulting formula is true or false. So there is no state space database manipulation. That is, roughly half of tableau expansions don't touch state state space database. Otherwise, generated state space are transferred or stored into the database. But even if random generation is assumed, 1/3 of transitions are store in local CPU. Since we are using clocked system, most common transitions are transition to itself. Because of these reasons, communication in this parallel verification is smaller than computation, which is good for parallel processing.

We also observe number of stored state is 2/3 of total state space 1422 which is sum of total import states. Current hash function simply assigns raondom number for each state, but still we have some state space locality. It is possible that each processor generates all possible state equally. In this case, all processor may have all state space. We don't know the source of the locality now. If we have good locality with state space, we can verify larger specification in the same machines.

If we check the subterm database, each processor contains the same number of subterms. This means all subterms are used in all processor. In this example, number of subterm is 10 times smaller than number of state. This means generations of subterm was done in the early stage of computation, which avoids access conflicts on the subterm server. This is because, in this example, most of the state is a combination of the state of the philosopher, and number of combination is much larger than each philosopher's state. This type if interleaving is common characteristics of parallel program, so we expect the same situation.

Here is the total packet count and retransmission (Table 5).

CPU	type	send count	resend count
1	state	2666	6148
2	state	1430	5509
3	state	2512	7312
1	RPC	228	0
2	RPC	146	0
3	RPC	236	0

Figure 5: **総パケット数と再送信数**

Number of retransmission is 10% of transmission. But in state stream, there are many retransmission, because we sent large number of packet without acknowledge. We expect that this problem is fixed by tuning of the Datagram library. In this experiments we use switching hub and full duplex ethernet, so there is no packet conflict. But Unix kernel may drop packets. In case of RPC, communications are synchronous and it is programed for response. But in case of state stream, we need throughput so there are many packet losses.

To see the speed up effect, we measure 3 dining philosopher problem with 10 processors (Table 5). In this case we use Linda library, since Datagram library is not debugged enough for 10 processors.

We have good speed up effect in 10 processor case (Fig. 6), although this problem is too small for 10 processor. We also see concentrating access on Linda packet server is not a serious problem in 10 processor case.

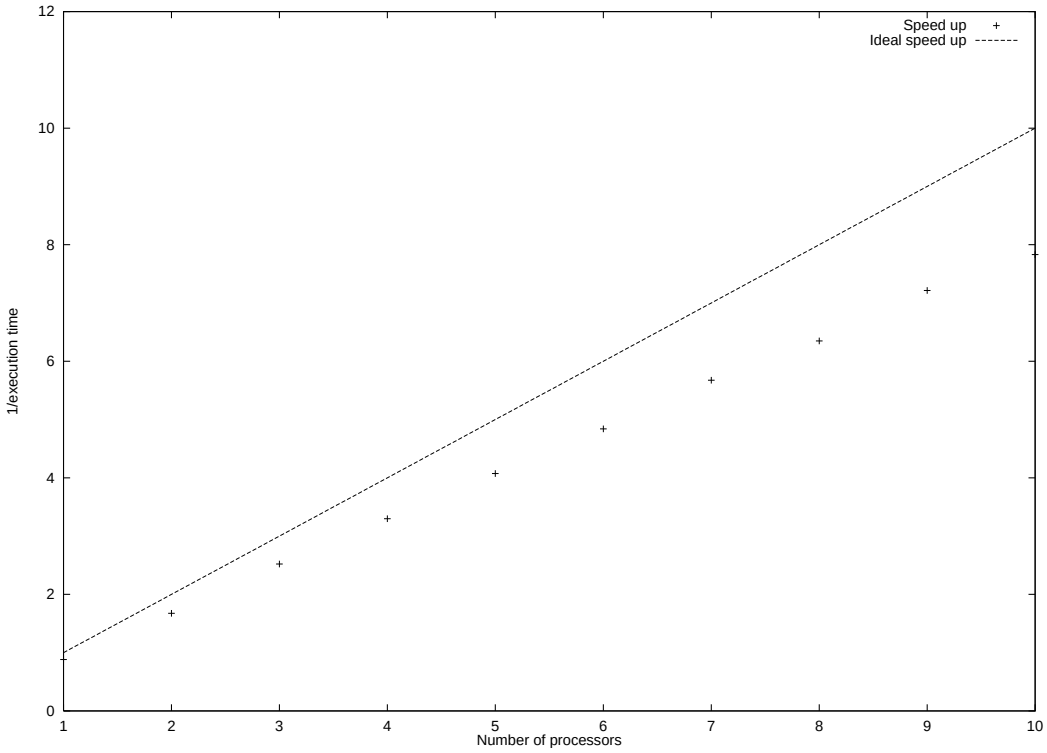

Figure 6: Speed up

6 Comparison

As a parallel verification system, BDD based model checking is known [3]. In ICOT, there is an implementation of LTTL tableau expansion in a parallel logic programming language GHC. All these implementation uses large shared memory for state database. But in our method state space is distributed among processors. In Fujitsu parallel machine AP1000 (cluster of Sparc station), there is a parallel implementation of derivation tree generation. The difference is that our method represents state space as a set of small Binary Subterm Tree. We also use Datagram library which makes it possible to use inexpensive PC Unix clusters without special hardware.

ITL tableau requires no eventuality checking because we have the end of time. So specification of fairness is impossible in ITL. But loop checking is expensive on distributed state space, this is an advantage from the view point of fast verification. In case of LTTL, there is no generation of new subterm during tableau expansion, but there are new subterms in ITL. We may generate possible subterms beforehand but since subterm management is cheap, we don't think it is necessary.

Several methods of Combination of sequential Prolog and distributed communication are known. The first one is Delta Prolog which is essentially Prolog with Remote Procedure Call. Sicstus's Linda package uses TCP stream. There is also a combination of PVM and Prolog. In case of PVM, communications are TCP stream and it is dynamically reconfigured during computation. Our

Datagram library may provides easy complete connection type communication, without centralized server.

UDP's advantages are;

- faster than TCP Stream
- No centralized server is required
- No reconfiguration of data stream is required
- Communication primitives are Prolog stream, no special predicates are necessary

But it also has disadvantages

- Complex library supporting Blocking/Deblocking, retransmission
- Throughput may decrease by retransmission
- Controlling Throughput/Response is difficult

In current library, timing of retransmission is done by program level, which makes our program complex. This should be done by library level.

In some sense, we implemented many-to-many TCP stream in terms of Datagram. Since we use sliding window, computation complexity is not so different with original TCP. But since we control sliding window in user level, we have advantages;

- it does not requires huge memory in Unix kernel
- No need of select call
- No need of system call to know communication states.
- We can control each processor's packet buffer.

But it is easy to generate large number of retransmission.

If we use synchronous communication in state stream, we can solve retransmission problem, but it reduces throughput. Once a processor becomes waiting state, other processor may wait also. So there may be cascading effects. But we put large number of packet to achieve higher throughput, it makes higher packet loss rate. We don't know how much throughput is required in case of large number of processors. More experiments are required in this area.

7 Conclusion

In this paper, we describes design, implementation and early experiments of parallel ITL verification.

This system still requires fine tuning and debugging, but early results is very good. If we can develop good state space division, we can achieve not only speed up, but larger scale verification.

References

[1] R.E. Bryant. "graph-based algorithms for boolean function manipulation". *IEEE Trans. Computer*, Vol. C-35, No. 8, pp. 667–691, Aug. 1986.

[2] Masahiro Fujita and Shinji Kono. Synthesis of Contrllers from Interval Temporal Loigc Specification. In *International Conference on Computer Design*, October 1993.

[3] K.L. McMillan. "symbolic model checking: An approach to the state explosion problem". Technical Report CMU-CS-92-131, Carnegie Mellon University, May 1992.

[4] B.C. Moszkowski and Z. Manna. Temporal logic as a programming language. In *Proc. of Parallel Computing 83*, Sept. 1983.

CPU	CPU time (sec)	States	Subterm	State tranmissions	export	import
1	725.896	945	71	6479	493	596
2	843.71	937	71	8306	382	335
3	713.055	907	71	6324	547	491
single	7496.414	1365	60	21075	-	-

Table 1: 5 dining philosophers using Datagram Library

CPUs	(sec)									
1	11.345									
2	6.758	5.178								
3	3.358	3.752	4.79							
4	3.978	0.89	5.7	1.56						
5	2.451	2.711	1.739	2.328	3.046					
6	2.134	1.282	3.715	0.114	4.066	1.085				
7	1.42	1.886	1.935	2.189	1.502	1.386	2.02			
8	1.828	1.663	1.325	0.698	2.407	1.095	1.887	1.697		
9	1.431	2.154	1.746	0.673	1.172	1.45	1.406	0.559	1.884	
10	1.733	0.867	1.494	0.374	2.598	0.545	2.294	0.398	1.907	0.563

Table 2: 3 dining philosophers using Linda Library