View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Purdue E-Pubs

Purdue University

Purdue e-Pubs

Department of Computer Science Technical Reports

Department of Computer Science

1991

# A Parallel Algorithm for Computing Invariants of Petri Net Models

Dan C. Marinescu

Mike Beaven

Ryan Stansifer

## Report Number:

91-024

Marinescu, Dan C.; Beaven, Mike; and Stansifer, Ryan, "A Parallel Algorithm for Computing Invariants of Petri Net Models" (1991). *Department of Computer Science Technical Reports.* Paper 873.
https://docs.lib.purdue.edu/cstech/873

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

**A PARALLEL ALGORITHM FOR COMPUTING INVARIANTS OF PETRI NET MODELS**

Dan C. Marinescu
Mike Beaven
Ryan Stansifer

CSD-TR-91-024
March 1991
(Revised September 1991)

To appear in the Proceedings of Petri Nets and Performance Models, PNPM91, Melbourne, Australia, December, 1991, IEEE Press.

# A Parallel Algorithm for Computing Invariants of Petri Net Models*

Dan C. Marinescu
Dept of Computer Sciences
Purdue University
W Lafayette, IN 47907

Mike Beaven
Dept of Computer Sciences
Purdue University
W Lafayette, IN 47907

Ryan Stansifer †
Dept of Computer Sciences
Purdue University
W Lafayette, IN 47907

**Abstract**

*Parallel algorithms for the analysis of Petri net models are discussed in this paper. It is argued that the application of Petri nets in areas like Performance Modeling and Software Engineering lead to complex nets whose analysis can only be performed by exploiting the explicit parallelism in existing methods of analysis and by developing parallel analysis algorithms. The focus of this paper is the structural net analysis. A parallel algorithm for computing net invariants using a distributed memory multiprocessor system is presented. We discuss its implementation, and give preliminary performance measurements.*

## 1 Introduction

Petri nets are used as graphic languages for description and more recently for the analysis of complex systems in various areas of Computer Science, of Electrical and Industrial Engineering and other fields.

There are challenging issues in the representation of complex systems as net models. Different families of net models were proposed for different application, for example Stochastic Petri nets for performance models, Ada nets for analysis of concurrent programs written in Ada [11] and so on. After constructing the net model of a complex system, the next stage of the modeling methodology is to apply the general methods of the net theory for qualitative or quantitative analysis of the net model. Finally, the results of the net analysis need to be remapped into the original domain and the relevant properties of the system under investigation have to be determined.

An application of Petri nets can only be successful if this cycle is complete. The focus of this paper is the second stage of this process, namely the net analysis. The primary area of applications discussed is Software Engineering. The main theme is that the complexity of the systems studied and the level of modeling details required to investigate certain aspects of the systems, lead to extremely large and very complex net models.

There are three broad classes of methods for the analysis of Petri nets. The first class of method is based upon *homomorphic transformations*. A good reference which outlines the principles behind transformation and decomposition of nets, is the paper of Berthelot [3]. It should be noted that in general, the transformations are aimed at reducing the complexity of a net, while preserving some properties which are relevant for the specified type of analysis being considered. For example, a set of rewriting rules for Petri net models of concurrent projections are discussed in [1]. These rules are tailored to preserve timing properties required by a critical path analysis of a concurrent program.

A second class of methods for net analysis are based upon *structural analysis*. Structural analysis, or the study of invariants, attempts to isolate from a set, subnets with special properties. For example, a place invariant of a net is a subnet, which preserves the number of tokens. Place invariants can be used for static deadlock detection as pointed out by Murata [11]. There are also important applications of transition invariants in modeling of logic programs [12] and Horn clauses [7]. The same algorithm can be used to compute both place invariants and transition invariants.

A third class of net analysis methods are based upon the *reachability analysis*. Detection of a dead marking, for example, can be done through reachability analysis. A dead marking corresponds to a deadlock state, no transition can fire.

Consider for example the application of Petri nets in Software Engineering of Real-Time Systems. The approach discussed in [2], is to translate concurrent programs into colored Petri nets and then to study synchronization anomalies and timing properties of the programs using structural analysis, as well as critical path analysis of the net models.

The size of the programs of interest makes this problem very challenging. For example the control program running on a communication switch of Bell Northern Research consists of about $5.10^6$ lines of code written in a high level real-time programming language. Using a translation technique like the one presented in [13], the incidence matrix and ultimately the net model of the concurrent program can be ob-

*This research was supported, in part, by a grant from the Software Engineering Research Center at Purdue University, a National Science Foundation Industry/University Cooperative Research Center (NSF Grant No. ECD-8913133), by the Strategic Defense Initiative through ARO grants DAAG03-86K-0106 and DAAL03-90-0107, and by NATO grant 891107.

†The author's present address is Dept of Computer Science, University of North Texas, Denton, TX 76205

tained. The complexity of the analysis depends upon the objectives of the analysis. For example, when one is interested in determining the timing correctness of an embedded system, a critical path analysis of the net representation of the program needs to be performed [1]. But relatively accurate timing estimates can only be provided at the machine instruction level, hence it is conceivable that each machine instruction has to be mapped into a different transition in the initial program representations. Simple reduction rules can be applied to reduce the numbers of transitions in the net model of the program, yet a net model of the control program mentioned above could be expected to have $10^7$ to $10^9$ transitions.

A distinction must be made between the size of a net and the complexity of the analysis, because nets of modest size may lead to a very complex analysis. For example, consider the "exponential net" discussed in Section 4, which consists of $a$ transitions with $b$ places in the preset or the postset of every transition. The total number of minimum support place invariants for the net is $b^{a+1}$. Even for a small net with say $a = 10$ and $b = 3$ the number of minimum support $P$-invariants is close to 60,000 and the time and memory requirements to solve such a problem are substantial.

The Mflops rates and the memory requirements for the analysis of such nets, impose the use of parallel systems. Distributed memory multiprocessor systems provide the kind of computing resources required by the analysis of large and complex net models. For example, the INTEL Touchstone Delta system provides in excess of 10 GFlops as peak rate and 8 Gbytes of memory, but parallel methods for net analysis need to be developed. Such methods have to exploit the explicit and the implicit parallelism in net analysis. The explicit parallelism can be determined by identifying subnets which can be analyzed in parallel. For example, the critical path analysis of two subnets corresponding to two program modules which do not communicate, can be carried out in parallel. The implicit parallelism can only be exploited by designing parallel algorithms which take a net as a whole and distributes in some fashion, the data and the computations among the processing elements of a distributed memory system.

The paper is organized as follows. A discussion of structural analysis, of its applications and an algorithm to compute minimum support $P$-invariants, are covered in Section 2. Section 3 presents a parallel algorithm to compute invariants for a distributed memory system and discusses its communication complexity. The implementation of the algorithm on an iPSC/i860 system and preliminary performance data are covered in Section 4.

# 2 An algorithm to compute invariants of Petri nets

A number of definitions and some of the notation for Petri nets and related concepts are introduced in this section. These will be used in the discussion of the algorithm for computing $P$-invariants. $P$-invariants and $T$-invariants are dual concepts. For this reason only algorithms to compute $P$-invariants are presented.

## 2.1 Definitions

A number of definitions and some of the notation for Petri nets and related concepts are introduced in this section. These will be used in the discussion of the algorithm for computing $P$-invariants.

**Definition 1** The quadruple $N = (S, T, F, W)$ is a Petri net when the following conditions hold:

- $S$ is the set of places,
- $T$ is the set of transitions,
- $S \cap T = \emptyset$,
- $F \subseteq (S \times T) \cup (T \times S)$ is the incidence relation,
- $S \neq \emptyset$ and $T \neq \emptyset$
- $W : F \rightarrow \mathbf{N}$, is the arc weighting function.

For any matrix, $M$, let $M_k$ be the $k$-th row of $M$ and $m_{ij}$ the element in the $i$-th row and $j$-th column of $M$. For any vector $J$, we use the notation $J(k)$ for the $k$-th element of $J$.

**Definition 2** Given orderings of $T = \{t_1, \ldots, t_m\}$ and $S = \{p_1, \ldots, p_n\}$ and defining two new functions $W^+(i,j) = W(t_i, p_j)$ and $W^-(j,i) = W(p_j, t_i)$, then the matrix $A$, an $n \times m$ matrix of integers with $a_{ij} = W^+(i,j) - W^-(j,i)$, is the *incidence matrix* of $P$.

**Definition 3** The *invariance matrix* is a matrix with $n$ columns. Initially, it is the identity matrix and at the end of the computation will have one row per $P$-invariant.

**Definition 4** If $J \cdot A = 0$, the $n$-vector $J$ is called an *$P$-invariant* of $P$.

**Definition 5** $P_J = \{p \in S \mid J(p) \neq 0\}$ is called the *support of $J$*.

**Definition 6** An $P$-invariant $J$ of $N$, which is not identical to 0, is *non-negative* if $J(p) \geq 0$ for all $p \in P$.

**Definition 7** The support $P_J$ of an $P$-invariant $J$ is said to be *minimal* if for any $P$-invariant $J'$, $P_{J'} \subseteq P_J$ implies that $P_{J'} = \emptyset$ or $P_{J'} = P_J$.

**Definition 8** Let $M_1$ and $M_2$ be $m_1 \times m_2$ and $m_1 \times m_3$ matrices. The augmented matrix $M = (M_1 \mid M_2)$ is the $m_1 \times (m_2 + m_3)$ matrix obtained by concatenating $M_1$ and $M_2$ such that the first $m_1$ columns of $M$ are identical to the corresponding columns of $M_1$ and the last $m_2$ columns of $M$ are identical to the corresponding columns of of $M_2$.

### 2.2 A sequential algorithm to compute $P$-invariants

This algorithm for computing $P$-invariants, is based upon an algorithm by Farkas [6] to solve a set of linear equations over the ring of integers. This algorithm may be looked upon as a graph rewriting algorithm in which places are created and deleted, while preserving the set of $P$-invariants of the original graph. Each new place accumulates the portion of the original graph which it comprises by means of an invariance matrix which augments the incidence matrix. For each step in the algorithm, we take a transition with a nonempty neighbor set and delete all of these neighboring places while adding a new place for each pairing of an incoming neighbor with an outgoing neighbor. This process continues until all the transitions have empty neighbor sets.

In the following description, for any $n \times m$ matrix, $M$, and length $m$ row vector, $J$, let $M + J$ be the $(n+1) \times m$ matrix formed by adding $J$ as the $(n+1)$-st row. Similarly, $M - i$ will represent the matrix formed by removing the $i$-th row of $M$. Both of these notations will be extended to sets of rows in the obvious manner. These will be the mechanisms by which we add or delete places which are represented by rows of the incidence and invariance matrices.

The initial step of the algorithm is to create an augmented matrix, consisting of the initial incidence and invariance matrices. This matrix is denoted $C^{(0)} = (B^{(0)} \mid A^{(0)}) = (I_n \mid A)$, where $B^{(0)}$ is the initial invariance matrix (equivalent to $I_n$, the $n \times n$ identity matrix), and $A^{(0)}$ is the initial incidence matrix (equivalent to $A$). $A_j^{(i-1)}$(i) is the $i$-th element of the row vector which is the $j$-th row of the incidence matrix A after iteration (i-1).

The following family of functions,

$$f_i : S \times S \rightarrow \mathbb{N}^{n+m}$$

defined by

$$(j,k) \mapsto \frac{|J(n+i)|}{gcd(I(n+i), J(n+i))} I + \frac{|I(n+i)|}{gcd(I(n+i), J(n+i))} J$$

where $J = C_k^{(i-1)}$ and $I = C_j^{(i-1)}$, are used in the statement of the algorithm. The function $f_i$ combines the rows of $C^{(i-1)}$ associated with places $p_j$ and $p_k$ to create the row representing the new place which is the merger of the places in the pair. Now, for each transition $t_i \in \{t_1, \ldots, t_m\}$ in turn, we compute the sets

$$\begin{aligned} S_+ &= \{p_j \in S \mid A_j^{(i-1)}(i) > 0\} \\ S_- &= \{p_j \in S \mid A_j^{(i-1)}(i) < 0\} \\ H &= \{f_i(k,j) \mid (p_k, p_j) \in S_+ \times S_-\} \end{aligned}$$

The $i$-th step of the algorithm then reduces to

$$C^{(i)} = (B^{(i)} \mid A^{(i)}) = C^{(i-1)} + H - (S_+ \cup S_-)$$

This algorithm is also described in a paper by Martínez and Silva [10]. In this paper an estimate of the rank of a matrix is used to eliminate non-minimal support invariants. A proof is given that the test is correct. Another test of eliminating non-minimal support invariants is found in GreatSPN [4]. In this test, each new row created at the $i$-th iteration of the algorithm is compared with existing rows and is deleted if its support covers the support of an existing row. The support of a place invariant $J_1$ is said *to cover* that of $J_2$ iff all non-zero components of $J_2$ are found in $J_1$. $J_1$ may have some additional non-zero elements. This test is critical for the efficiency of the algorithm. Only in some pathological cases, like in the case of the exponential net discussed in Section 4, where all invariants are minimum support invariants and the number of invariants grows exponentially with the size of the net.

## 3 A Parallel Algorithm for Computing Invariants

In this section a parallel algorithm to compute $P$-invariants using a distributed memory multiprocessor system, is introduced. First an overview of distributed memory multiprocessor systems and the message passing programming paradigm is discussed. An outline of the algorithm is then presented. Finally an analysis of the communication complexity is given.

### 3.1 Distributed memory multiprocessor systems

A distributed memory multiprocessor system consists of a set of nodes and an interconnection network. A node consists of a processor, main memory and possibly, a co-processor. Most of the commercially available systems use either a mesh or a hypercube interconnection network. For example, the iPSC/i860 system at CSC (Concurrent Supercomputer Consortium) is a 64 node hypercube. Each numeric node uses an i860 processor for numerical computations, has 16 Mbytes of memory and has a co-processor responsible for message routing. The system is controlled by a System Resource Manager running on a front-end or host.

An application consists of a host program and one or more node programs. The host program is responsible for acquiring the necessary resources for the application, e.g., for obtaining a sub-cube, for loading in the nodes of the sub-cube the corresponding node program and data, and for releasing the sub-cube upon completion. Node programs perform computation upon local data and communicate with one another through messages, using communication primitives like *send* and *receive*.

The main difficulty in designing algorithms for distributed memory multiprocessors, is in partitioning the data and the computations to

(a) minimize the communication between nodes,

(b) ensure a balanced load distribution among the nodes.

Communication among nodes is rather expensive, the time to send a short message (say 1 byte) between two adjacent nodes is roughly the time to perform $10^3$

to $10^4$ floating point operations on a node, and this justifies the requirement (a) above.

The algorithm in the next section is based upon the SPMD (Same Program Multiple Data) paradigm. All nodes execute the same program, but on different data.

### 3.2 A parallel algorithm to compute $P$-invariants

The algorithm for computing $P$-invariants in parallel follows closely the sequential algorithm described in Section 2.2. Ownership of the incidence matrix $A$ and the invariants matrix $B$ is distributed cyclically among the $N$ processors as shown in Figure 1. The $i$-th row of the invariants matrix is owned by processor $i \bmod N$; the $j$-th column of the incidence matrix is owned by processor $j \bmod N$.

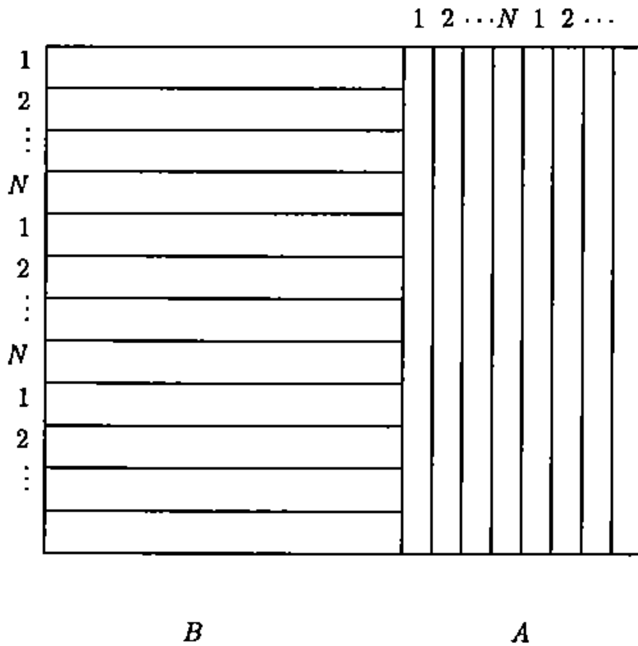


Figure 1: The cyclic data distribution for computing invariants with $N$ processors.

The initial step is the same as in the sequential algorithm with each processor storing the rows and columns it is to own. The processor owning the next column to be eliminated then broadcasts to each other processor a vector consisting of the list of rows with negative and positive elements in that column, $S_+$ and $S_-$, respectively. Each processor then bundles the rows it owns into groups based on which other processors need them to create the new rows they are to own and sends the bundled rows to each processor the proper grouping. It then receives from some (possibly all) of the other the rows it needs to create the new rows it is to own. After combining these rows and storing the new row, it deletes the rows used it owns which where in the original broadcast vector. The processors then all synchronize and continue with the next transition until all transitions have been examined.

The pseudo code describing the main computation follows. Each processor executes the same code. The variable iam holds the unique processor number, 0 through $N-1$, of the node.

```
for (col=1 ; col<=num_cols ; col++) {
    if (iam == col%N) {
        compute_transform_vector();
        broadcast_transform_vector();
    } else {
        receive_transform_vector();
    }
    update_incidence_matrix();
    distribute_invariant_matrix_rows();
    receive_invariant_matrix_rows();
    combine_invariant_matrix_rows();
    delete_old_invariant_matrix_rows();
    gsych();
}
```

In the initialization phase, one of the nodes (the leader) reads the incidence matrix and broadcasts it to all other nodes. Each node retains only the columns of the incidence matrix it owns. In the termination phase, the leader informs the host when all computations have completed.

During the phase where new rows are created, each new row is subjected to the non-minimal support test used in GreatSPN. This test, however, is only carried out for those preexisting rows which are stored on the local processor. This may allow some non-minimal rows to remain, but reduces the communications complexity. The set of preexisting rows includes not only the rows permanently stored on the local processor, but those rows temporarily stored there for the creation of the new rows during the current iteration of the algorithm. In addition, the rows which are permanently stored on the local processor are tested against the rows received from the other processors since this adds very little extra computation and no extra communications.

### 3.3 On the Analysis of the Algorithm

Numerical problems involving manipulation of sparse matrices pose challenging questions to the design of a parallel algorithm [5]. It is extremely difficult to achieve dynamic load balance for parallel solvers of sparse linear systems. The strategy adopted in the algorithm presented in this paper is to assign to each node an approximately equal number of columns of the incidence matrix and rows of the invariants matrix. If the number of transitions of the net $m \gg P$ with $P$ the number of nodes in the sub-cube assigned to the problem, then the data mapping strategy used guarantees that

$$M_{max} - M_{min} \leq 1$$

with $M_{max}$ and $M_{min}$ the maximum, and respectively the minimum number of columns of the incidence matrix assigned to a node.

Yet there is no guarantee of dynamic load balance. Let $\delta_i$ be the number of new invariants generated when processing column $i$ of the incidence matrix $A$,

$$\delta_i = (|S_+|) * (|S_-|) - ((|S_+|) + (|S_-|))$$

where $|S_+|$ is the number of positive elements in column $i$ of $A$, and $|S_-|$ is the number of negative elements in column $i$ of $A$. Clearly $\delta_i$ varies from node to node, depending on the structure of column $i$ and the different nodes will have different computational loads. Another major difficulty in designing efficient algorithms for sparse equations solvers, is the requirement for global synchronization [8]. In this particular algorithm, a global synchronization is required at the end of each of the $m$ steps.

Following the arguments presented in [8] and [9] in this type of algorithm, the relationship between the number of events $E$ and the total number of threads of control $N$,

$$E = O(N^2).$$

An event is informally defined as an interruption of the flow of control in a thread for communication. Simple arguments show that in this case the maximum speedup and the optimal number of processors depend upon the total amount of computations required by a sequential algorithm, called $W(1)$ and the constant amount of computations associated with a single event, $\theta$. If $\alpha = \frac{W(1)}{\theta}$ then the maximum speedup attainable is shown in Table 1.

| $\alpha$ | $10^2$ | $10^4$ | $10^6$ |
|---|---|---|---|
| $S_{max}$ | 5 | 50 | 500 |
| $N_{opt}$ | 10 | 100 | 1000 |

**Table** 1. Maximum speedup and the optimal number of threads of control function of $\alpha$.

This brief analysis suggests that the algorithm will lead to significant speed up only for very large problems ($W(1)$ large) and for systems with efficient communication primitives and fast communication hardware ($\theta$ small).

# 4 Experimental Results

## 4.1 Implementation and methodology

The results reported in this paper were obtained using two different iPSC/i860 hypercubes, a 16 node machine at Purdue and a 64 node machine at the California Institute of Technology. The available memory is 16 Mbyte per node for both systems. Two Green Hills C compilers are used. The i386/1.8.4 is used to compile the host program and the i860/1.8.5b for the node program.

The iPSC routines from the C system libraries used in the implementation are csend, crecv, and gsync. The library routine csend causes the sending node to wait until the message is sent, not until the message is received by the target node. The library routine crecv is a blocking receive which waits until a message of a specified type is received, and gsync which causes all nodes which execute it to wait until all other nodes in the cube have called gsync.

The measurements are carried out using the mclock system call which provides the elapsed time since the cube was booted with one millisecond accuracy. Each measurement was repeated several times and the timing results were within 5 percent of each other. Yet no statistical analysis of the timing results was performed.

## 4.2 Performance of the parallel algorithm

The performance of the parallel algorithm for computing $P$-invariants was studied in two extreme cases. In the first case called *linear Petri net* every transition is connected to one input place and one output place. The output place serves as the input place for the next transition. Such a Petri net has exactly one $P$-invariant. The other extreme case called *exponential Petri net* consists of $t$ transitions and each transition has $p$ places in its preset and postset. The net has $p^{t+1}$ invariants.

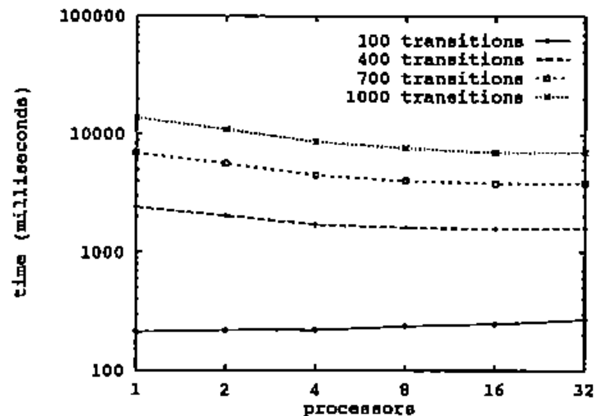


Figure 2: A log–log plot of the timing data for linear Petri nets with one $P$-invariant each.

The data shown in Figure 2 is for linear Petri nets with different numbers of transitions. Along the $x$-axis, in logarithmic scale, are the number of processors. The $y$-axis represents time, in milliseconds, also in logarithmic scale. In Figure 3 the data for exponential Petri nets is presented. Even though these nets have few transitions, many new rows of the invariance matrix are created at each step of the algorithm. In fact, for large $n$ many processors are required to even have enough available memory to run the program. Note that that any test for minimal support $P$-invariants would only add to the running time of the algorithm. This is because the test will eliminate no rows regardless of the algorithm, since all the $P$-invariants of the exponential net are minimal.

Yet a more realistic case is shown in Figure 4. The data shown in Figure 4 is for a Petri net generated by the VERT project [2] for the following simple 19 line Ada program.

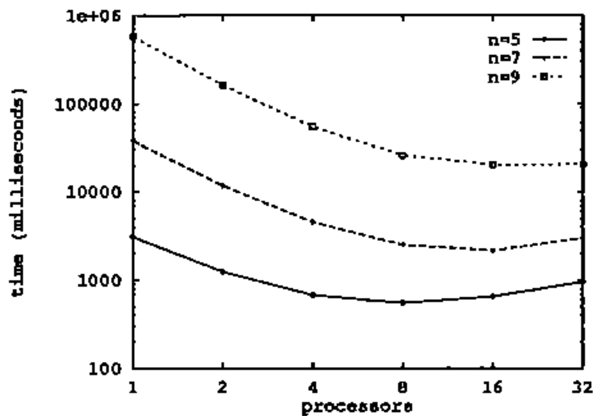


Figure 3: A log-log plot of the timing data for Petri nets with $2^n$ $P$-invariants each.

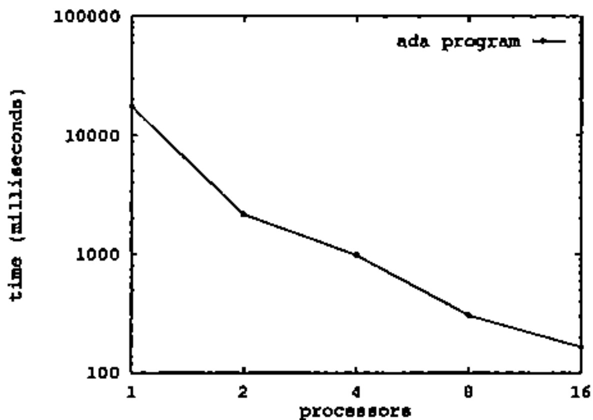


Figure 4: A log-log plot of the timing data for a Petri net model of an Ada program.

```
procedure P is
    task type TT;
    type PtrTT is access TT;
    X : PtrTT;
    Y : PtrTT;
    procedure S is
        task T is
            entry Z;
        end T;
        task body T is
        begin
            accept Z;
        end T;
    begin
        T.Z;
    end S;
    task body TT is begin S1; S2; end TT;
begin
    X := new TT;
    P.S;
    S3;
    Y := new TT;
    P.S;
end P;
```

This program is translated into the colored Petri net shown in Figure 5. The Petri net model for the program has 18 places and 14 transitions. The algorithm finds 392 $P$-invariants. The steeply decreasing slope suggests that a parallel algorithm for computing $P$-invariants may be more useful in the "typical" case than in the two pathological cases presented above.

As expected from the analysis in Section 3.3, the algorithm does not perform well for "small" cases. For example, in the case of a linear Petri net (Figure 2) the optimum number of processors for the $10^3$ transition case is probably 16. Yet the speedup does not increase significantly when the number of processors is increased to 16.

In the exponential case the parallel algorithm performs better. The optimum number of processors for $2^5$ case is 8, it increases to 16 for the $2^7$ case and to 32 for the $2^9$ case. This shows clearly that the algorithm is suited only for large nets. A comparison shows that in most cases the parallel algorithms outperforms the sequential algorithm for the same net only when the number of processors allocated to the problem exceeds $P = 8$. This indicates that further optimization of the parallel algorithm is necessary. The test for minimal support $P$-invariants will provide additional information to determine the range of problem when the parallel algorithm could prove its usefulness.

The analysis of the Petri net mode of an Ada program (Figure 4) shows a good performance of the parallel algorithm, better than its performance on the two pathological cases, the linear and the exponential nets. A benchmarking methodology for testing the analysis algorithms for Petri nets is necessary and much work is needed in that area. In fact the Ada net shows better than linear speedup for the two processor cases. A plausible explanation is that the node memory management is rather inefficient in performing "garbage collection." When a single node is used, the repeated

Figure 5: Petri net model of procedure P.

storage allocation and deallocation lead to large overhead. When the computation is distributed over two nodes, each node uses less storage and the overhead for storage management decreases substantially. The node operating systems are rather primitive and measurements to confirm this explanation could not be carried out at this stage.

The minimal support invariant test of the algorithm introduced in this paper requires that each node performs the test locally upon the invariants it owns and then broadcasts the information about which invariants are to be deleted so that all other nodes can update the columns of incidence matrix they own. The effectiveness of the test depends upon the net being analyzed. In the extreme case of the exponential net, all invariants are minimal support invariants and therefore there is no pay off for the additional computations and communication required by the test. This is also the case of several other nets obtained by translating Ada programs with few synchronization points. For example, an Ada program with two tasks and four entry points has a nearly tri-diagonal incidence matrix and only few nonminimal support invariants are generated. The execution time increases from 264 msec (without the test) to 313 msec (with the test) on a dimension 3 subcube (8 nodes) and from 438 msec (without the test) to 536 msec (with the test) on a dimension 2 subcube (4 nodes). But in other cases, the test is beneficial. For example, in the case of the Ada program presented in this section, the execution time decreased from 305 msec (without the test) to 47 msec (with the test) when running on a dimension 3 subcube. Further investigation of the effectiveness of the test is underway.

## 5 Conclusions

A parallel algorithm for computing $P$ invariants is introduced in this paper. Its implementation on a distributed memory machine is presented. The analysis of the communication complexity of the algorithm shows that one could expect good performance for very large cases. The experimental results confirm this. A test case using a Petri net produced as a result of translating an Ada program shows a higher speedup than the two pathological cases, a linear and an exponential net.

## References

[1] Beaven, M., Marinescu, D. C., and Stansifer R., "Critical path analysis of concurrent Ada programs using colored Petri nets, in *Proceedings of 24th IEEE International Symposium on Circuits and Systems*, IEEE Press, 1991, (in press).

[2] Beaven, M., Elmore, B., Marinescu, D. C., and Stansifer R., "VERT—Verification of real-time programs," in *Proceedings of COMPSAC 91*, IEEE Press, 1991, (in press).

[3] Berthelot, G., "Transformation and decomposition of nets," in *Lecture Notes in Computer Science*, volume 254, Springer Verlag, 1987, pages 359–376.

[4] Chiola, G., "A graphical Petri net tool for performance analysis," in *Proc. of 3-rd Symp. on Modeling Techniques and Performance Evaluation*, AFCET, Paris, 1987.

[5] Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J. and Walter, D., *Solving problems on concurrent processors*, Prentice Hall, 1988.

[6] Farkas, J., "Theorie der einfachen Ungleichungen," *Journal für reine und angewandte Mathematik*, volume 124, 1902, pages 1–17.

[7] Lin, C., Chandhury, A., Whinston, A. B. and Marinescu, D. C., "Logical inference of Horn clauses in Petri net models," CSD-TR-91-031, Purdue University, 1991.

[8] Marinescu, D. C., and Rice, J. R., "Synchronization and load imbalance effects in distributed memory multiprocessor systems", in *Concurrency: Practice and Experience.*, Willy, 1991, (in press).

[9] Marinescu, D. C. and Rice, J. R. "On single parameter characterization of parallelism," in *Proc. Frontiers 90 Conf. on Massively Parallel Computation*, IEEE Press, 1990, pages 235–238.

[10] Martínez, J. and Silva, M., "A simple and fast algorithm to obtain all invariants of a generalized Petri net," in *Application and Theory of Petri Nets*, edited by W. Reisig and C. Girault, Springer-Verlag, 1982, pages 301–310.

[11] Murata, T., Shenker, B., and Shatz, S.M., "Detection of Ada static deadlock using Petri net invariants," *IEEE Transactions on Software Engineering*, volume 15, No 3, 1989, pages 314–326.

[12] Murata, T. and Zhang, D., "A predicate-transition net model for parallel interpretation of logic programs," *IEEE Transactions on Software Engineering*, volume 14, No 4, 1988, pages 481–497.

[13] Stansifer, R. and Marinescu. D. C., "Petri net models of concurrent Ada programs," in *Microelectronics and Reliability*, Pergamon Press, Volume 31, No 4, 1991 pages 577–594.

[14] Stansifer, R. and Marinescu, D. C., "Colored Petri net models of concurrent programs," in *33rd Midwest Symposium on Circuits and Systems*, IEEE Press, New York, 1991, pages 766–770.