

## Parallel Graph Generation Algorithms for Shared and Distributed Memory Machines

S.C. Allmaier<sup>a</sup> and S. Dalibor<sup>a</sup> and D. Kreische<sup>a</sup>

<sup>a</sup>Computer Science Department III, University of Erlangen–Nürnberg,  
Martensstr. 3, D–91058 Erlangen, Germany.  
{*snallmai* | *dalibor* | *ddkreisc*}@informatik.uni-erlangen.de

In this paper we give an overview and a comparison of two parallel algorithms for the state space generation in stochastic modeling on common classes of multiprocessors. In this context state space generation simply means constructing a graph, which usually gets extremely large. On shared memory machines, the key problem for a parallelization is the implementation of a shared data structure which enables efficient concurrent access for retrieving the nodes of the graph. In our realization this search structure is a B-tree with special locking mechanisms assigned, leading to significant speedups. For distributed memory machines a dynamically adaptive partitioning strategy to distribute the state space onto different processors together with load balancing mechanisms is implemented. Thus sequentially not manageable problem sizes can be solved by combining the main memories of clustered workstations.

### 1. INTRODUCTION

When constructing models of complex systems it is essential to represent all the global states of the system by considering only the local states, i.e. the states of its subcomponents. This leads to a very compact description of the information contained in the model. However, to get detailed answers about the occurrence and interdependencies of events and the behavior of the entire system, it is necessary that all the global states are available to the analysis process. This means the *state space* of the model — consisting of the set of all global states and the possible transitions between them — has to be generated. Mathematically the state space can be formalized as a graph.

Complex systems and their timing behavior may be modeled in a compact way using *Generalized Stochastic Petri Nets* (GSPNs) [11]. GSPNs are popular in many areas of science and engineering like the performance and reliability analysis of computers, the verification of programs and network protocols and the control and optimization of business and manufacturing processes.

When using GSPNs, local states of the system are represented explicitly by so called *places*. The global state of the system is then given by the distribution of resources — which are called *tokens* — on the places. Such a global state is called a *marking* of the GSPN. Global state changes are represented by redistributing some of the tokens according to specific rules given by the Petri net definition (with these rules synchronization, concurrency and timing of events can be modeled easily). Therefore only the initial marking has to be defined by the modeler and all *reachable* global states can be deduced from that initial state automatically. This process corresponds to the state space generation mentioned above. In the Petri net formalism, the state space is called *reachability graph*; its nodes are the reachable markings of the GSPN, its edges are the transitions between them.

Since realistic models tend to lead to very large global state spaces (state space explosion

problem [3]) the state space generation consumes a major part of the overall computation time of the analysis process. Additionally, the resulting state spaces are often too large to fit into the main memory of a single workstation or PC.

Although parallelization of the state space generation process is a non-trivial problem, it is an appropriate means to reduce time spent on analysis; additionally the size of the models that can be solved is enlarged by orders of magnitude. For shared memory machines, the time to build up a model's reachability graph can be shortened drastically by exploiting the common access of several CPUs to large amounts of memory. In the case of networked workstations or PCs, a parallel algorithm for distributed memory multiprocessors can pool the physically distributed memories so that very large state spaces can be analyzed without the need for special hardware or expensive supercomputers.

In this paper we present new parallel algorithms for the generation of the reachability graph of GSPNs for shared and distributed memory multiprocessors and report on significant speedups. Our implementations enable us to analyze Petri nets with large state spaces both on a Convex SPP 1600, a Sun Enterprise Server 4000 and on a cluster of DEC Alpha machines.

## 2. PARALLEL REACHABILITY GRAPH GENERATION

The reachability graph generation is an iterative process [4] which starts with the initial marking only and leads to a dynamically growing data structure, namely the reachability graph  $R$ , whose final size is not known in advance. New markings are generated out of an already existing marking — i.e. a node of  $R$  — by finding all of its successor markings, which are uniquely defined by the GSPN model.

The algorithm resembles those of other state-space enumeration techniques, e.g. the branch-and-bound methods used for the solution of combinatorial problems such as computer chess and the traveling salesman problem. However, the main difference is that the result of the computation is not a single value (such as the minimum path length in the traveling salesman problem or a position evaluation in a game of strategy) but the entire state space itself. Therefore no cutoff can be performed, i.e. the complete state space has to be generated.

To ensure that one marking is not inserted several times into  $R$ , the set of already existing reachability graph nodes has to be searched every time a potentially new node is investigated. For being able to generate large state spaces in acceptable time, it is crucial to maintain a search data structure  $D$  that allows very fast lookups and updates. Our algorithms use a balanced search tree for this purpose.

Alternatively a hash table could have been used instead. The advantages of a hash table would have been an easier implementation and much more simple data synchronization mechanisms on a shared memory machine by just locking the hash table entry that is accessed, as well as savings in memory since in contrast to search trees no left and right child pointers have to be maintained.

However, due to the fact that the size and shape of the state space are not known until it is entirely generated, there would have been difficulties in defining the size of the table and the hash function itself. We cannot simply allocate a hash table as big as possible as it may be done in computer chess algorithms, since we also have to store the edges of  $R$  which occupy a share of memory that is not known a priori. So we would have to implement dynamic hashing which abates the first advantage. Additionally in computer chess it is possible to simply neglect collisions in the hash table whereas we have to keep *all* the nodes of the graph. So we would additionally have to have at least a linear list to our disposal in each hash table entry to handle collisions which abates the second advantage.

The third dynamically changing data structure in our algorithm is a set of all the unprocessed markings  $S$ . In our implementation we organized  $S$  as a stack. It contains all reachability graph nodes whose successor markings have not been generated yet.

The basic (sequential) algorithm of the reachability graph generation in pseudo-code is given as:

```

1 procedure generate_reachability_graph
2   initial marking  $m_0$ 
3   reachability graph  $R = \emptyset$ 
4   search data structure  $D = \emptyset$ 
5   stack of unprocessed markings  $S = \emptyset$ 
6   begin
7     add node  $m_0$  to  $R$ 
8     insert  $m_0$  into  $D$ 
9     push  $m_0$  onto  $S$ 
10    while ( $S \neq \emptyset$ )
11       $m_i \leftarrow \text{pop}(S)$ 
12      for each successor marking  $m_j$  to  $m_i$ 
13        if ( $m_j \notin D$ )
14          add node  $m_j$  to  $R$ 
15          insert  $m_j$  into  $D$ 
16          push  $m_j$  onto  $S$ 
17        endif
18      add edge  $m_i \rightarrow m_j$  to  $R$ 
19    endfor
20  endwhile
21 end generate_state_space

```

For reasons of efficiency and scalability, the parallelization has to be “homogeneous” in the sense that each CPU performs all steps in processing one marking  $m_i$  concurrently with all other CPUs.

## 2.1. Shared Memory Multiprocessor Solution

Shared memory multiprocessors can be exploited best by allowing global concurrent access for all the CPUs to the entire reachability graph and search structure. This means the state space is not partitioned onto the processors.

### 2.1.1. Problems

- To guarantee consistency of the dynamically growing data structures, elaborated synchronization methods have to be applied.
- To achieve a high degree of parallelism, the share of global data structures that is locked for mutual exclusive access has to be kept small.
- On the other hand synchronization operations may not occur too frequently because they usually are time-consuming.
- As the number of nodes and edges of  $R$  cannot be determined in advance, an efficient dynamic memory allocation method suitable for parallel execution has to be applied.

Probably due to these problems we could not find any shared memory solutions for parallel reachability graph generation of Petri nets in literature up to now.

### 2.1.2. Our Solution

Data consistency with a high degree of parallelism is achieved in our algorithm by using B-trees [7,5] as search structure  $D$  whereby the markings are ordered and serve as search key:

- B-trees are automatically balanced, which reduces the time spent for searching to  $O(\log(n))$  (where  $n$  is the number of reachability graph nodes).
- B-trees allow concurrent access for reading and writing by protecting B-tree nodes with semaphore variables [5].
- Several markings are grouped together to one B-tree node which is the entity that is locked. This way synchronization operations are reduced.
- By the use of a modified routine for balancing the tree [2,7], each CPU locks at most two B-tree nodes at a time (which are a parent and its child node). While the tree is traversed, the locks move down from the root towards the leaves of the tree. Since searching always starts at the root of the B-tree, the probability of colliding memory accesses decreases with the depth of the locked nodes. Thus a high degree of parallelism is achieved.

As  $R$  is always accessed via the search structure, no data protection has to be applied there. See [2,1] for a detailed explanation of the algorithm (which would be beyond the scope of this comparison).

As far as we know, all system libraries for dynamic memory allocation are single-threaded, meaning memory management operations are performed in mutual exclusion. This would severely degrade the performance of our algorithm since small memory blocks (one for each node and edge of  $R$ ) are allocated in high frequency. Therefore we had to implement our own memory management on top of the system libraries: each CPU requests memory in large portions from the system and administers them locally, thus eliminating potential conflicts with other CPUs.

## 2.2. Distributed Memory Multiprocessor Solution

### 2.2.1. Problems

- As in the case of shared memory, each processor has to generate a part of the reachability graph. However, since memory is distributed, each CPU has access only to the limited part of the state space which is located in its private memory.
- As a consequence, a part of the set of all potential states has to be assigned to each processor a priori — i.e. *before* the generation starts — by defining a suitable mapping function on the markings (*partitioning of the state space*). This means the state space is split in advance so that for each newly generated marking the responsible processor is uniquely determinable and the marking can be sent to it.
- Nevertheless the partitioning of the state space has to distribute nodes and edges of the accruing reachability graph equally among the CPUs (*load balancing*). If no proper load balancing strategy is implemented, reachability graph generation for many models would have to be aborted prematurely due to a lack of memory on a subset of the CPUs (while other CPUs' memories remain unused).
- For efficiency reasons, the partitioning function should map connected regions of  $R$  (i.e. groups of nodes which have edges mainly among themselves) onto one CPU. This way the amount of communication could be reduced: successor markings would not have to be sent to other processors (*principle of locality*).

Two distributed memory algorithms are known to us, namely [9] and [6]. In this approaches static hash functions are used to map markings onto processors. This way it cannot be guaranteed that the memories of all the processors are utilized best: the number of reachability graph nodes per processor may differ because of an imperfect hashing function. Additionally, edges that are automatically mapped together with the corresponding nodes may be badly distributed.

### 2.2.2. Our Solution

In our approach, dynamic load balancing was implemented to ensure an equal distribution of the state space among the processors. However, for our purposes memory consumption instead of computation time is regarded as load. The introduction of dynamic repartitioning is the main difference between our approach and existing distributed memory parallelizations of the state space generation of GSPNs [9,6].

A master/slave-scheme is used: on each CPU a slave process is started which performs the actual work of creating new reachability graph nodes; on one processor an additional master process for administering and controlling the slaves is initiated.

The slaves permanently generate new markings. According to the partitioning function, these markings are either stored in an output buffer (pending to be sent to another slave) or are handled locally.

To ensure that the state space is equally distributed among the processors, dynamic load balancing is implemented whereby a chain topology of the processors is assumed:

- The entire set of possible markings is ordered and each slave is assigned a continuous (not necessarily equal sized) range of potential reachability graph nodes.
- A load balancing phase is initiated by the master if the memory utilization of one processor differs more than  $n\%$  from the average memory utilization of the other processors ( $n$  can be chosen by the user). In our experience, 10% is a default value that produces best results for all the models we investigated.
- The master tells each slave how much memory — measured in bytes — it has to give to its two neighboring processors.
- The search structure  $D$  of each processor again is a balanced tree; each slave may send those nodes of  $R$  whose corresponding markings are minimal/maximal with respect to the given order to its left/right neighboring processor. Hereby the balance of the tree is easily maintained since the nodes are sent and received in an ordered way. The sending processor measures the number of bytes while giving nodes and edges to its neighbor, thus guaranteeing the precomputed distribution.
- It is not possible that a processor has too few data to give to its neighbor since in this case it gets data from its other neighbor. All the slaves start sending concurrently as far as possible.
- Afterwards the slaves send their new maximal and minimal markings to the master that recomputes the resulting range of potential markings of each slave — i.e. the new partitioning function — out of this information and broadcasts it to all the slaves.

A slave runs idle if all the markings generated locally or received from the other slaves are processed. At this point it flushes its output buffers and broadcasts an idle message. The other slaves flush the output buffer which corresponds to the idle slave, thus delivering new work for the idle process. The algorithm terminates if all slave processes become idle and no markings are left in the output buffers.

This way our algorithm takes into account all the problems mentioned in Section 2.2.1 except for the locality problem which may be seen as indwelled in the nice condensed way information is expressed in the high level GSPN model: there are no methods known to anticipate the shape of the state space in advance.

## 3. RESULTS

We report on the results of performance measurements for both the shared and distributed memory approach. The intention is to show the benefits of the two different approaches from

the viewpoint of the modeler: the shared memory parallelization proves to be able to utilize a supercomputer efficiently resulting in significant savings of time, whereas the distributed memory approach is a more cost-effective but time consuming way of making model sizes manageable which could not be handled with just one workstation alone.

Extensive experiments with a broad variety of different GSPN models were performed; we describe the measurements for one of them in detail: the GSPN [10] models failure and repair activities in a fault-tolerant multiprocessor system. It consists of 13 places (local states) and 25 GSPN transitions. Measurements were done with different values for the initial number of modeled CPUs in the system (i.e. the number of tokens representing CPUs in the initial marking of the GSPN): modeling 26 CPUs results in a reachability graph with 160,000 nodes and 1,500,000 edges and 28 modeled CPUs lead to a reachability graph with 200,000 nodes and 1,900,000 edges.

### 3.1. Shared Memory

**Convex implementation:** The algorithm was implemented on a Convex SPP1600 parallel computer which provides 2GB of global virtually shared memory for a cluster of 8 HP PA-RISC 7200 CPUs. Within this cluster memory is accessed in a uniform way via a crossbar switch. The implementation was done using the ConvexOS CPS thread libraries.

**POSIX implementation:** The Convex program has been ported to use the standardized POSIX thread interface; measurements have been conducted on a Sun Enterprise server 4000 with 8 UltraSparc-1 CPUs and 2GB of main memory. CPUs and memories are connected in a symmetric way via a crossbar switch and a packet switching bus.

### 3.2. Distributed Memory

For the measurements we used a cluster of 6 DEC 2100/500MP server machines, each equipped with a DEC Alpha CPU and 128MB of main memory. The machines are connected by an FDDI ring. The parallelization was done within the PVM [8] programming environment.

### 3.3. Experimental Results

Fig. 1 shows the speedups of the parallel algorithms for two different model sizes. It can be seen that the shared memory version works very well leading to a nearly optimal linear speedup on both the Convex and the Sun machine. The speedup of the distributed memory version is comparatively poor but yet some time is saved since the parallelization prevents the machines from swapping. Without swapping effects the algorithm gains no speedup as the communication overhead of this irregularly structured problem is too high and the models that can be used for speedup measurements are too small.

Fig. 2 shows the computation times for the GSPN modeling 28 CPUs measured on different architectures. On the DEC cluster even for this relatively small models swapping effects degrade the computation time in the monoprocessor case; the parallel performance is worse than that of the shared memory machines (due to the significantly slower communication hardware and the message passing library overhead), but not to an unbearable extent.

Fig. 3 shows that the main goal of the distributed memory parallelization has been reached: it depicts the maximum size of the reachability graph of the GSPN model that can be computed using a certain number of processors of the DEC cluster. While in the sequential (monoprocessor) case the GSPN with more than 28 modeled CPUs (i.e.  $R$  consisting of 200,000 nodes and 1,900,000 edges) cannot be evaluated due to a lack of memory, it is possible to construct the reachability graph with up to 38 modeled CPUs (i.e. 600,000 nodes and 5.3 million edges in  $R$ ) if 6 computers of the DEC cluster are used.

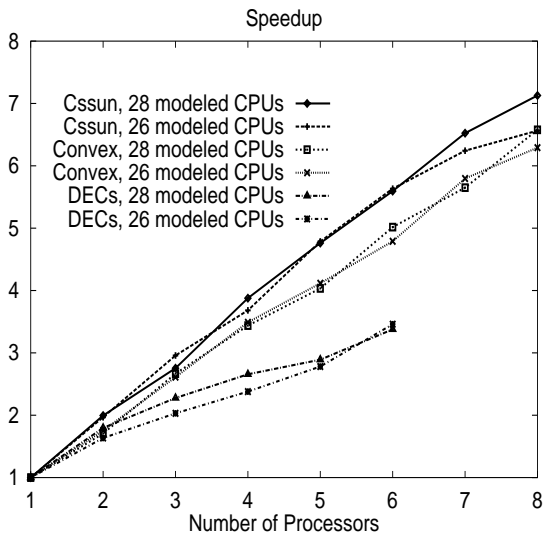


Figure 1. Speedups for different reachability graph sizes on different architectures (shared and distributed memory).

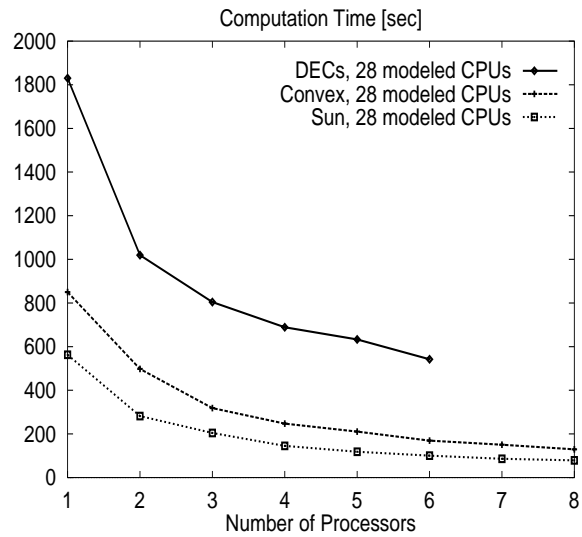


Figure 2. Computation times on different architectures (shared and distributed memory).

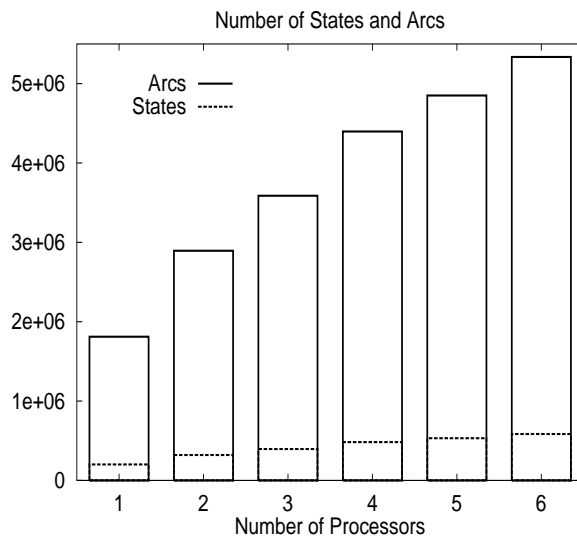


Figure 3. Largest analyzable model sizes for different numbers of processors of the DEC cluster.

#### 4. SUMMARY AND OUTLOOK

We presented a parallel solution for the problem of generating reachability graphs of GSPNs both for shared and distributed memory multiprocessors. In the case of shared memory multiprocessors we can conclude that it is possible to shorten analysis times for Petri nets with huge state spaces significantly if suitable parallelization methods are applied. For workstation clusters our parallelization enabled us to analyze GSPN models whose reachability graphs would be by far too large to be generated using a single machine of the cluster.

The conclusions above also give rise to a comparative evaluation of the different multiprocessor architectures we used for our implementations: The algorithm for distributed memory machines has to be much more complex to compensate for the missing global data access. Therefore, no real speedup can be gained compared to the shared memory case and analysis times are longer. On the other hand, the distributed memory solution enables the analysis of large models on commodity hardware (as workstation clusters with high-bandwidth connections become more and more widespread), whereas the shared memory implementation needs an expensive high-end server or a supercomputer when large models are to be evaluated. If such a machine is available, parallel generation of the reachability graph may accelerate the analysis process considerably.

We plan to do more measurements on other parallel machines in the near future. Parallel implementations of the GSPN analysis steps following reachability graph generation and the integration of our programs into an automated tool for complete GSPN analysis are underway.

#### REFERENCES

1. S. Allmaier and G. Horton. Parallel shared-memory state-space exploration in stochastic modeling. In G. Bilardi, A. Ferreira, R. Lüling, and J. Rolim, editors, *Solving Irregularly Structured Problems in Parallel*. Springer, 1997. LNCS 1253.
2. S. Allmaier, M. Kowarschik, and G. Horton. State space construction and steady-state solution of GSPNs on a shared-memory multiprocessor. In *Proc. IEEE Int. Workshop Petri Nets and Performance Models*, St. Malo, France, 1997. IEEE Comp. Soc. Press.
3. G. Balbo. On the success of stochastic Petri nets. In *Proc. IEEE Int. Workshop on Petri Nets and Performance Models*, pages 2–9, Durham, NC, 1995. IEEE Comp. Soc. Press.
4. G. Balbo, G. Chiola, G. Franceschinis, and G. Molinar Roet. On the efficient construction of the tangible reachability graph of generalized stochastic Petri nets. In *Proc. IEEE Int. Workshop on Petri Nets and Performance Models*, pages 136–145, Madison, WI, USA, 1987.
5. R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9:1–21, 1977.
6. G. Ciardo, J. Gluckman, and D. Nicol. Distributed state-space generation of discrete-state stochastic models. Technical Report 198233, ICASE, NASA Langley Research Center, Hampton, VA, 1995.
7. D. Comer. The ubiquitous B-tree. *Computing Surveys*, 11(2):121–137, 1979.
8. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, 1994.
9. P. Marenzoni, S. Caselli, and G. Conte. Analysis of large gspn models: A distributed solution tool. In *Proc. IEEE Int. Workshop Petri Nets and Performance Models*, St. Malo, France, 1997. IEEE Comp. Soc. Press. To appear.
10. M. Ajmone Marsan, G. Balbo, and G. Conte. *Performance models of multiprocessor systems*. MIT Press, 1986.
11. M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with generalized stochastic Petri nets*. Wiley, Series in Parallel Computing, 1995.