State Space Construction and Steady–State Solution of GSPNs on a Shared–Memory Multiprocessor

S. C. Allmaier, M. Kowarschik and G. Horton

Lehrstuhl für Rechnerstrukturen (IMMD III),

Universität Erlangen-Nürnberg,

Martensstr. 3, 91058 Erlangen, Germany.

{snallmai | mskowars | graham}@informatik.uni-erlangen.de

Abstract

A common approach for the quantitative analysis of a generalized stochastic Petri net (GSPN) is to generate its entire state space and then solve the corresponding continuous–time Markov chain (CTMC) numerically. This analysis often suffers from two major problems: the state space explosion and the stiffness of the CTMC. In this paper we present parallel algorithms for shared–memory machines that attempt to alleviate both of these difficulties: the large main memory capacity of a multiprocessor can be utilized and long computation times are reduced by efficient parallelization. The algorithms comprise both CTMC construction and numerical steady–state solution. We give experimental results obtained with a Convex SPP1600 shared–memory multiprocessor that show the behavior of the algorithms and the parallel speedups obtained.

1 Introduction

1.1 Background

The steady–state behavior of a GSPN [18] can be analyzed either by discrete–event simulation or by generating the state space of the underlying stochastic process, namely a continuous–time Markov chain (CTMC), and then solving this CTMC numerically. If some events occur substantially less frequently than others, simulation can become time–consuming and numerical solution becomes the method of choice. However there are two principal potential problems in the numerical analysis of GSPNs [1, 25]: state space explosion and CTMC stiffness.

The state space of the underlying stochastic process of a GSPN is often very large. The memory capacity and performance of a typical monoprocessor workstation may therefore impose severe practical restrictions on the set of tractable models.

There are two ways to deal with the state space explosion problem: the first is avoidance by the use of state space reduction methods [22, 23, 17, 2, 20, 4]. However, these may require more or less restrictive net properties for their application. The second is to tolerate the size of the state space by either limiting the computation to data relevant for the required output quantities [21], reducing complexity at the net level [19], using sparse storage techniques based on the existence of model invariants [7] or by parallelization [5, 7], which is also the topic of this paper.

Stiffness in CTMCs can adversely affect the convergence behavior of standard numerical solution methods like Gauss–Seidel [24, 14], leading to unacceptably long computation times. Stiffness often occurs when the rates of events in the system differ strongly in magnitude, as is usually the case with dependability models. The solution method used here, however, has proven to be quite robust with respect to stiffness.

1.2 Our Contribution

In this paper we present new parallel shared–memory algorithms for GSPN analysis both for the construction of the CTMC and for its numerical solution. We developed our algorithms for shared–memory architectures since these have become widely available as mainframe supercomputers and as high-end workstation server machines, and also because they offer a convenient programming model for the concurrent processing of large unstructured data sets.

Our state space generation can be performed on any GSPN, as defined by the well–known modeling tool SPNP [8], without any structural restrictions. In addition, we allow the timed transitions to be of phase type, rather than just exponentially distributed. All memory policies of the non–exponential timed transitions [16] are allowed, and the automatic generation of the state space is transparent to the user.

In contrast to distributed memory algorithms for state space generation, we do not need to partition the state space explicitly, because global memory gives each processor access to the entire data set. Our data structures allow a high degree of parallelism while guaranteeing consistency. Since global data is concurrently updated and read very frequently, suitable synchronization mechanisms are crucial to parallel efficiency. Our synchronization strategies are adapted from parallel file system maintenance algorithms [3].

For the numerical steady–state solution of the CTMC we developed a shared–memory parallelization of the multi–level algorithm presented in [15]. This is also a general–purpose algorithm, which does not assume any special model characteristics, and which is especially efficient for stiff CTMCs.

We present performance measurements of a thread–based implementation of our methods on a Convex SPP1600 shared–memory multiprocessor with 4 Gbytes of main memory. Results with a selection of test cases indicate that our algorithms parallelize well and achieve speedups which are essentially problem–independent.

The organization of the paper is as follows: Section 2 gives a short description of the state space construction algorithm and the basic concepts of its shared–memory parallelization. Section 3 describes the steady–state solution of the CTMC using the parallelized multi–level method. In Section 4 we report on the performance of our parallel algorithms using a Convex SPP1600. Section 5 ends with a conclusion and a survey of the tasks remaining for the future.

2 Parallel State Space Construction

Steady–state analysis of a GSPN can be done in three steps [6], the first two of which will be described in this section:

1. Generation of the extended reachability graph (ERG) which defines the underlying stochastic process. This is a semi–Markov process (SMP) whose sojourn times are either exponentially distributed or constant zero.

2. Reduction of the SMP to a CTMC by eliminating all states with zero sojourn time — these are the so–called *vanishing* states in which one or more immediate transitions are enabled.

3. Solution of the resulting system of linear equations to obtain the steady–state probabilities.

2.1 Parallel ERG Generation

2.1.1 The Basic Algorithm

The ERG is a directed graph, whose states correspond to the markings of the GSPN and whose weighted arcs represent the rates or probabilities with which state changes occur. Our task is to create the entire ERG, starting from the initial marking. Algorithm seq_generateERG() in Figure 1 performs the (sequential) ERG generation. New markings are generated by firing each of the transitions that are enabled in the current marking. The processing of a new marking leads to the creation of an arc in the ERG. The algorithm terminates when there are no more markings to investigate, at which point the entire (finite) ERG has been generated. The variable m refers to the current marking and s is the corresponding state in the ERG. We refer to the source state of an arc as the *parent* of its destination state in this section.

2.1.2 Data Structures

Shared memory parallelization concepts are based on the idea that all data may be accessed by all the parallel processes or threads, as they all share the same address space. Therefore it is crucial for the efficiency of shared–memory parallelizations that the design of the global data structures allows concurrent manipulation. So we will first have to consider the data

```
ALGORITHM seq_generateERG()

m ← initial marking
create new state s corresponding to m in ERG
push s onto stack
WHILE ( stack ≠ empty )
    get parent state: s_p ← pop(stack)
    FOR (all successor markings m of s_p)
        look for state s corresponding to m in search structure
        IF ( not found s ? )
        THEN
            create new state s corresponding to m in ERG
            insert s into search structure
            push s onto stack
        ENDIF
        create new arc in ERG with source s_p and destination s
    ENDFOR
ENDWHILE
```

Figure 1: Algorithm for ERG generation

structures used by the ERG generation algorithm of Figure 1 before we can describe the design decisions and synchronization mechanisms we apply in our parallelization (Section 2.1.3).

There are three main data structures that are used by algorithm seq_generateERG(): a stack, the ERG itself and a search structure.

Stack. Newly generated states are stored in a stack until their turn to be processed.

ERG. Arcs are linked to the data structures of their destination, rather than their source states, since neither the CTMC generation nor the numerical solution needs to access outgoing arcs. This storage scheme is very important for avoiding both synchronization within the ERG structure (as we will see in Section 2.1.3) and unnecessary memory consumption.

Search structure. The search structure is used to retrieve previously created states which are identified by the marking they represent. To do this efficiently — especially for large state spaces — the search structure has to fulfill two criteria: short search times and low memory consumption. To achieve this, hash tables require an accurate a priori estimate of minimal table size and also a hash function that distributes the markings equally within the table. For GSPNs, however, neither of these objectives seems feasible. Therefore we decided to use a balanced search tree: this provides search times that are logarithmic and a memory consumption that is linear in the number of states.

2.1.3 Parallelization

We now describe the parallelization of the main loop of algorithm seq_generateERG() of Figure 1. Since during one pass through the loop of the algorithm the time spent on generating the new markings, looking for a marking in the search structure and updating both the ERG and the search structure turned out to be similar, all of these operations must be done in parallel. All threads carry out the main loop of algorithm seq_generateERG() concurrently, but each operates on a different marking at any one time.

Parallelization Problem. This parallelization approach leads to the problem of maintaining data

consistency: each thread must be able at any time to decide if the marking it is processing corresponds to an already created state or not, which means that it must always have access to the most recent update of all global data.

Parallel ERG generation algorithms for distributed-memory machines are reported in [7] and [5]. There, this problem is solved by partitioning the state space onto the processes in advance. Each process accesses only its own part of the ERG and the search structure, from which it automatically follows that it uses the most recent update. A process has to send all the markings it generates that do not belong to its partition to the appropriate process, which may cause a large communication overhead. In both cases cited, it was found that for some models a load and memory imbalance occurred, which was caused by an inappropriate partitioning that mapped unequal numbers of markings to processors. Therefore, the algorithm of [7] requires the user to apply knowledge of the GSPN model in order to specify the hash function that maps markings onto processes.

By contrast, a shared-memory architecture allows each thread access to the entire ERG and search data structure, thus obviating the need for partitioning and — as a consequence — for such heuristics. There are no load balancing problems to be concerned with, but data structures have to be designed which allow efficient concurrent access, along with suitable *synchronization* methods for maintaining data consistency.

Synchronization. Synchronization is needed when concurrent reading and writing of the global data may lead to inconsistencies. Consistency can be achieved by protecting the data by the use of *locking variables*. If data is only accessed when locked — be it for reading or for writing — then this access is done under mutual exclusion. To maximize the degree of parallelism, it is therefore important that the volume and frequency of data locking be as small as possible.
Locking schemes on B-trees.

Since all the threads concurrently read from and insert into the search tree and since these operations are done very frequently — reading is probably performed several times for each ERG arc and an insertion takes place for each ERG state — efficient synchronization strategies on the search tree are crucial for the performance of the algorithm. We found an efficient way to maintain the balance of the tree by allowing concurrent access through the use of B-trees, which are described in detail in [10].

A B-tree is said to be of *order* σ if the maximum number of keys that are contained in one B-tree node is 2σ. A node containing 2σ keys is called *full*. The search keys of one node are ordered smallest key first. Each key can have a left child which is a B-tree node containing only smaller keys. The last (largest) key in a node may have a right-child-node with larger keys. Here the GSPN markings — represented by integer vectors — serve as search keys, whereby the lexicographical ordering is imposed.

Searching is performed in the usual manner: comparing the current key in the tree with the one that is being looked for and moving down the tree according to the results of the comparisons. An unsuccessful search always leads to an insertion into a leaf node. Insertion into a full node causes the node to split into two parts, promoting one key up to the parent node which may lead to the splitting of the parent node again and so on recursively. Splitting might therefore propagate up to the root. Note that the tree is automatically balanced, because the tree height can only increase when the root is split.

In the parallel context, B-trees allow various synchronization schemes for maintaining data consistency. In [3], several locking methods are described, assigning one or more locking variables to each B-tree node. In our experiments we observed that using more than one locking variable causes an overhead that is not justified since the operations on each state are not very time-consuming. The easiest way to avoid data inconsistencies is to lock each node encountered on the way down the tree in the course of the search. Since non-full nodes serve as barriers for the back-propagation of splittings, all locks in the upper portion of the tree can be released when a non-full node is encountered [3]. However, using this approach, each thread may hold several locks simultaneously. Moreover, the locked nodes are often located in the upper part of the tree where they are most likely to cause a bottleneck. Therefore, and since we do not know a priori if an insertion will actually take place, we have developed another method, adapted from [12], that we call *splitting-in-advance*.

Our B-tree-nodes are allowed to contain at most $2\sigma + 1$ keys. On the way down the B-tree each full node is split immediately, regardless of whether an insertion will take place or not. Back-propagation does not occur, since parent nodes can never be full. Therefore a thread holds at most one lock at a time. The lock moves down the tree as the search proceeds. This keeps access conflicts between threads to a minimum, allowing high concurrency of the search tree processing. Figure 2 shows the insertion of key 17 in a B-tree
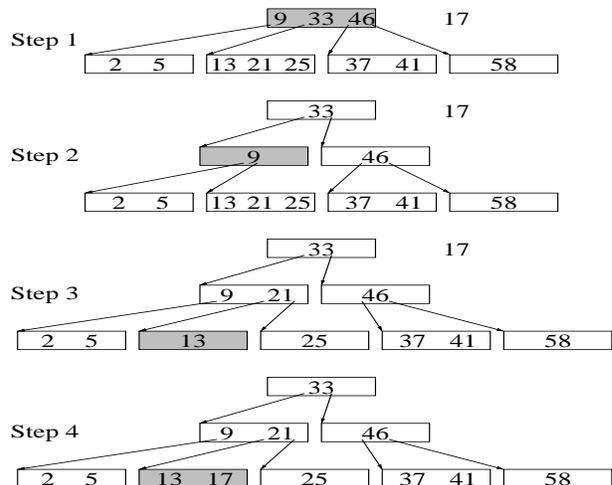

Figure 2: Splitting-in-advance in a B-tree

of order $\sigma = 1$ using splitting–in–advance. Locked nodes are shown as shaded boxes. As the root node is full, it is split in advance in Step 1. The lock can be released immediately after the left child of key 33 has been locked (Step 2). The encountered leaf–node is again split in advance, releasing the lock of its parent (Step 3). Key 17 can then be inserted appropriately in Step 4 without the danger of back–propagation.

Using sparse storage methods — i.e. organize B-tree nodes as linked lists rather than arrays, which in general will not be full — the data organization is similar to that of binary trees, whereby B–trees consume at most one more byte per ERG state than binary trees (for details see [10]).

Locking on the stack.

Since a single shared stack would be a considerable bottleneck, we additionally assign a private stack to each thread. Each thread uses its private stack, only pushing new markings onto the shared stack if the latter's depth drops below the number of threads. A thread only pops markings from the shared stack if its private stack is empty. In this manner load imbalance is avoided. The shared stack is accessed concurrently and thus has to be protected by a locking variable. On the other hand, as reading and writing the variable representing the stack depth are atomic operations, no locking is needed when accessing it.

The overall locking scheme.

Our algorithm uses one locking variable per B–tree node and one for the shared stack. No barriers are needed during the ERG generation, allowing a maximum degree of independence between the threads.

Each thread holds at most one lock at a time. The data structures that require synchronization are the shared stack and the search tree. Because arcs are linked to the data structures of their destination, rather than their source states, no additional synchronization is needed for the ERG data structure. This can be seen from an analysis of algorithm `seq_generateERG()`, which reveals:

- ERG states are always accessed via the search structure.
- It is the destination state that is looked up before a new arc is created (Figure 1).
- Other threads are prevented from accessing the destination state by holding the lock of its corresponding B–tree node until the arc is inserted (Figure 2).
- The arcs of the parent state are not manipulated.

Termination. Termination of the ERG generation algorithm is determined by the status of the global and private stacks. Each thread has a local termination flag which is readable for all the other threads. Termination testing is performed by each thread only when it encounters an empty shared stack (implying that its private stack is empty, too). The thread sets its termination flag and then tests all the other termination flags under mutual exclusion by holding the lock of the shared stack until the end of the termination test. If the test fails, the thread waits a random time before again trying to pop a marking from the shared stack.

2.2 Parallel Elimination of Vanishing States

The second step in the analysis of a GSPN is the elimination of vanishing states from the reachability graph in order to obtain a continuous-time Markov chain. There are two methods available: post–elimination and elimination on–the–fly:

1. *Post–elimination* first constructs the complete ERG and subsequently eliminates all vanishing states and arcs to obtain the CTMC. We implemented post–elimination first because it preserves the possibility of computing impulse rewards, which require the entire ERG [9].

2. *Elimination on–the–fly* means constructing the CTMC directly from the GSPN without permanently storing all the vanishing states. This method bypasses the creation of the ERG.

We now give a brief description of the two ways of constructing the CTMC in parallel. In Section 4 the two approaches are compared.

2.2.1 Post-Elimination of Vanishing States

The description of the overall locking scheme in Section 2.1.3 showed the importance of storing the arcs with (only) their destination state. The CTMC generation algorithm must be constructed to allow parallelism utilizing only this information. The parallel generation of the infinitesimal generator of the CTMC can be done by first partitioning the *tangible* ERG states — these are the states in which only timed transitions are enabled — equally onto the threads. Each thread then visits all its tangible states and eliminates all of the latter's incoming *vanishing arcs*, i.e. arcs that have a vanishing source state. The elimination of a
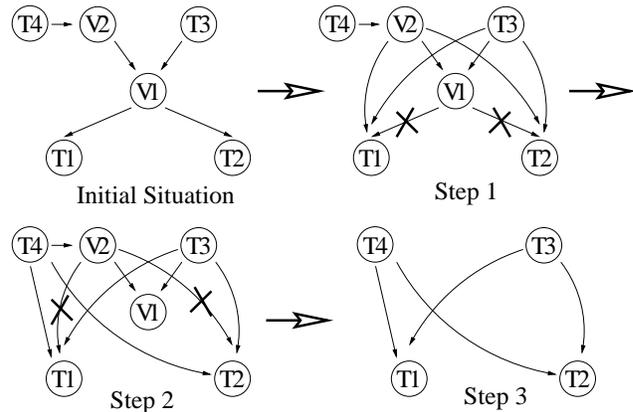

Figure 3: Parallel CTMC generation by eliminating vanishing states

vanishing arc is done by replacing it by newly–created incoming arcs connecting all the parent states of its source state with its destination state and performing the corresponding modifications to the rates [6]. When

chains of vanishing states are encountered, the elimination proceeds recursively in a breadth–first manner. Figure 3 shows the elimination of the vanishing states by two threads in parallel for a portion of an ERG. Vanishing states are denoted by 'V' and tangible ones by 'T'. Tangible state T1 is mapped onto Thread 1 and state T2 onto Thread 2. In Step 1, each thread replaces a vanishing arc by two new arcs in parallel. Since two of the new arcs are also vanishing, they are concurrently eliminated in Step 2 in the course of the breadth–first recursion. Since the only data that is modified are the incoming arcs of tangible states, and since the only data read concurrently are the vanishing arcs of vanishing states, no locking is needed at all. Synchronization is limited to one barrier, which forces the threads to wait until all the vanishing states with their arcs can be explicitly deleted (Step 3).

Vanishing loops in the state space of a GSPN are usually considered illegal and therefore lead to an error message. Vanishing loop detection is done separately by a partial depth–first search over the vanishing states before they are eliminated [7]. Its computation time can be neglected.

The advantage of post–elimination is that it generates the ERG. It has the consequent disadvantage of a higher amount of storage, since all vanishing markings have to be stored. In a typical GSPN the number of vanishing states can easily equal or exceed the number of tangible states; the storage requirements for this method could thus be double those of the on–the–fly algorithm.

2.2.2 Elimination on-the-fly

Elimination on–the–fly means that during the reachability graph generation vanishing states are not actually generated and inserted in the search structure, but are only temporarily stored until further exploration of the state space again encounters a tangible state. Then a link can be made directly between the tangible state from which the first vanishing marking was found to the latter tangible state. For a detailed description of the on–the–fly algorithm see [9] or [7]. The method needs only slight modification for parallel execution and is therefore not described further here.

In the parallel case each processor maintains a separate local stack for the temporary vanishing markings. There are no special problems to overcome when eliminating on–the–fly in parallel since all the computations concerning vanishing states can be done locally without need for any knowledge about the global state space. Since there is no synchronization or locking necessary for the vanishing markings, parallel elimination on–the–fly is very efficient on a multiprocessor.

The advantages of the on-the-fly approach are that it can save a considerable amount of memory, since only a handful of vanishing markings must be stored at any one time, and that it can be parallelized more efficiently. On the other hand it can be more expensive than the post–elimination scheme, since the processing of the same sequences of vanishing markings may have to be carried out several times over.

3 Steady–State Analysis Using the Multi–Level Method

The multi–level method [15] is an iterative algorithm for computing the steady–state probability distribution of a CTMC. It makes use of a system of CTMCs created by recursively coarsening the original CTMC. Its convergence behavior can be significantly better than that of traditional numerical solution methods like Gauss–Seidel/SOR when stiff or large CTMCs are solved [14]. Furthermore, the algorithm is well–suited to be parallelized on a shared–memory architecture, as we will show.

3.1 Sketch of the Multi–Level Method

The multi–level solution process consists of an *aggregation phase* and a subsequent *iteration phase*.

Aggregation phase. First, a hierarchical system of recursively coarsened CTMCs (*levels*) has to be generated, (Figure 4). This aggregation process terminates when a CTMC is generated that is small enough to be solved easily using a standard solution method. The aggregation maps groups of fine–level CTMC states to common coarse–level states, resulting in a new, smaller CTMC. The fine–level CTMC states that are aggregated to the same coarse–level CTMC state are referred to as the *child states* of the latter, which is referred to henceforth as the *parent state*.
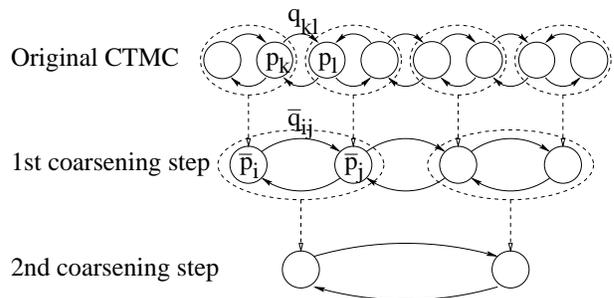

Figure 4: Aggregation of CTMCs

In order that the coarse level system has a solution in which coarse state probabilities are the sums of their children's probabilities, the coarse transition rates $\bar{q}_{ij}$ from state i to state j depend both on the fine–level transition rates q_{kl} and the fine–level steady–state solution probabilities p_k (or current approximations to these) according to the *aggregation matrix*, whereby $s_k \in \bar{s}_i$ denotes that coarse–level state $\bar{s}_i$ is the parent of fine–level state s_k:

$$\bar{q}_{ij} := \frac{\sum\limits_{s_k \in \bar{s}_i} p_k \sum\limits_{s_l \in \bar{s}_j} q_{kl}}{\sum\limits_{s_k \in \bar{s}_i} p_k}. \qquad (1)$$

Iteration phase. After the aggregation phase is finished, *multi–level iteration cycles* have to be performed on the original CTMC until its approximate solution has reached the required precision.

A single multi–level iteration on a finer CTMC of the hierarchy consists of five steps [15]:

1. *Pre–smooth* the current approximation to the solution of the CTMC by performing a few Gauss–Seidel iteration steps. This improves the relative probabilities in states with a common parent.
2. *Restrict* the smoothed fine–level solution to the next coarser level and compute new values for the coarse–level transition rates.
3. Recursively perform a *multi–level iteration* on that coarser level; on the coarsest level of the hierarchy, solve the CTMC exactly using a standard solution method. This obtains improved values for the probabilities of aggregates and thus for the sums of the probabilities of their children.
4. *Correct* the fine–level approximation by applying the relative changes in the coarse–level probabilities to their fine–level members.
5. (Optionally) *post–smooth* the corrected fine–level approximation using Gauss–Seidel.

3.2 Aggregation Algorithm

We have not yet considered parallel aggregation strategies, because when solving the CTMC sequentially, the major portion of the computation time is usually consumed by the iteration phase. The design of parallel aggregation strategies will be an essential part of our future work.

The aggregation strategy must fulfill the following two conditions:

- Only small groups of strongly coupled states, i.e. those which are connected by transitions with comparatively large rates, are aggregated.
- Two states are prevented from being aggregated if at least one of them is connected significantly more strongly with a third state.

In order to achieve this, the aggregation algorithm successively visits all arcs in the fine CTMC in descending order of rate. Our implementation therefore uses a sorted list of arcs to generate coarse–level CTMC states, which is processed sequentially. This strategy does not lend itself well to parallelization. We are presently trying to find an aggregation strategy that allows a high degree of parallelism while still fulfilling the above conditions.

Finally, we generate the arcs of the coarse CTMC. Whenever there is an arc connecting two fine–level CTMC states that are mapped to different parent states, a coarse–level arc connecting those two parent states has to be introduced. This part of the aggregation can be parallelized without difficulties.

3.3 Multi–Level Iteration

In this subsection we discuss some algorithmic aspects of the parallel multi–level method (see also [13]).

We use a global barrier to synchronize the threads following each of the five steps mentioned above in order to ensure that all threads are working on the same step at any time during the iteration phase.

Partitioning the CTMCs. Our algorithm partitions each CTMC of the hierarchy into as many sets of states as there are parallel threads. This partitioning can be completely arbitrary (see Figure 5) — we simply partition the index set — because the shared

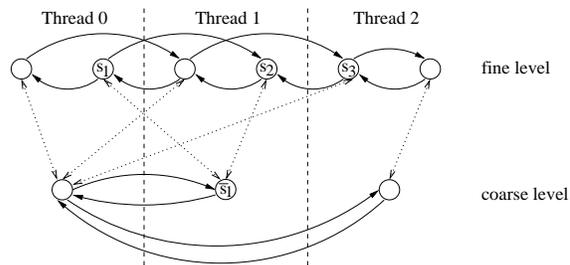memory architecture allows each thread to access the entire data of the hierarchy.

Figure 5: Partitioning the CTMCs

Smoothing the approximate solution. Our parallel Gauss–Seidel smoothing procedure (Steps 1 and 5 of the algorithm) implicitly reorders the unknowns so that each thread can process its partition in one sweep without needing synchronization. Retaining the sequential ordering of states would incur excessive idle times while waiting for value–updates done by other threads. This modification can affect the performance of the method, but greatly enhances its parallelism.

Restricting fine–level approximations. The restriction procedure (Step 2 of the algorithm) consists of two phases: the transfer of the pre–smoothed fine–level approximation to the next coarser level, and the re-computation of the coarse–level transition rates according to Equation (1). Our algorithm performs both of these steps entirely in parallel. In the solution transfer step, each thread sums the solution values of all children of each coarse–level state in its partition. These quantities represent the values of the denominator in Equation (1). In Figure 5 Thread 1 computes the restricted approximation for coarse–level state $\bar{s}_1$ by successively visiting its child states s_1 and s_2 and accumulating their current steady–state solutions. At the same time, the values of the numerators in Equation (1) can be determined by inspecting the incoming arcs of the child states of each coarse–level CTMC state. Second, after synchronization with a global barrier, the division can be performed in parallel.

Correcting fine–level approximations via interpolation. This step (Step 4 of the algorithm) does not impose any difficulties in parallelizing since there are no data dependencies within each level at all. The coarse–level correction is obtained by dividing the approximation resulting from the multi–level cycle on the coarse level by the approximation yielded by the previous restriction step. Each thread can then multiply the coarse correction back into the values of each of its fine–level states. In Figure 5 Thread 0 corrects the approximation of the fine–level state s_1 by reading the correction factor of the corresponding parent state $\bar{s}_1$ and then performing the multiplicative correction.

Normalization and error computation. Normalization to a probability vector and error computation both require a reduction and a broadcast operation which are easily realized with shared–memory using a lock in conjunction with a global barrier.

4 Experimental Results

4.1 Description of the Experiments

We implemented our algorithms on a Convex Exemplar SPP1600 multiprocessor with 4 Gbytes of main memory. This machine uses non–uniform memory access with physically distributed memory. In this sense the results are worst–case and the measurements should be representative for modern shared–memory machines. Our implementation used Convex–specific threads. We have also ported our implementation to a Sun Enterprise machine which uses POSIX threads. In all cases reported below, each thread is mapped to a different processor.

Our experiments use three scalable GSPN models:

- Model A is used to analyze the performance of a dynamic–priority operating system [11],
- Model B is adapted from [18] and describes a multiprocessor system with failures and repairs,
- Model C is taken from [26] and models a flexible manufacturing system.

In practice we can handle state spaces of more than 4 million states and 25 million arcs on a subcomplex with 2 GBytes of memory without needing data compression methods. The models used here are much smaller because other users had to be excluded whilst the experiments were done. This is because the SPP does not provide accurate user–dependent timer values and that context switches indirectly lead to severe delays owing to cache reloading. In our measurements we use the model sizes given in Table 1. With models of twice the size, one sequential experimental run would have taken about 45 hours.

		Model A	Model B	Model C
ERG	states	1,200,000	880,000	780,000
	arcs	2,500,000	3,200,000	2,100,000
CTMC	states	300,000	260,000	190,000
	arcs	1,600,000	2,300,000	1,400,000

Table 1: Used State Space Sizes for Different Models

In contrast to the ERG and CTMC generation, the multi–level solution shows a variance in execution time. This is caused by the non–deterministic ordering of the states of the CTMC when it is generated in parallel. The dependence of Gauss–Seidel on the ordering can affect the multi–level convergence rate. In addition, the number of levels may vary because of the heuristic nature of the aggregation strategy. Hence, all multi–level measurements are averaged over five runs. The values vary from the mean value by 7% on an average, whereby the maximum difference is 30%.

4.2 Measurements

Figure 6 shows the computation times needed for the different phases of our algorithms depending on the number of processors using Model C: the CTMC generation applying elimination on–the–fly, the ERG generation followed by the post–elimination of the vanishing states, the aggregation of the multi–level algorithm and finally the multi–level iteration. Figure 7 shows the corresponding speedup values.

In the monoprocessor versions used for these measurements, all locking operations were deleted and the B–tree order of the CTMC generation was optimized: $\sigma = 1$. Note that the good speedup for the state space generation emphasizes the efficiency of our locking mechanisms. Post–elimination of the vanishing states is not very time–consuming. For this model computation times and speedups for post and on–the–fly elimination do not differ very much. The aggregation phase of the multi–level algorithm is largely sequential and therefore no significant speedup is gained. The time for the aggregation is, however, non–negligible, and we are therefore currently working on a parallel version.

Figures 8 and 9 compare computation times and speedups for the CTMC generation using the post and on–the–fly elimination methods for Models A and B. Besides the advantage of saving storage (more than two thirds of the states are vanishing), for these models elimination on–the–fly gains better speedups than post–elimination. But as explained in Section 2.2.2, chains of vanishing markings may be traversed several times, which in these cases results in higher computation times for the elimination on–the–fly. For Model A, where the number of arcs per state is very small, the speedup compensates for this additional work so that elimination on–the–fly is faster on eight processors. A comparison with Figure 7 suggests that the speedups seem to be nearly model–independent.

In Figure 10 the total number of markings popped from the private stacks during the ERG generation is compared with the number of markings popped from the shared stack for Model A. The left bars correspond to the private stack and the right ones to the shared stack respectively. When eliminating vanishing states on–the–fly, the numbers are smaller because all the vanishing markings are kept on a separate, local stack and do not appear in this figure. The number of accesses to the shared stack when eliminating on–the–fly are so small, that they cannot be seen in the diagram. The figure shows that many conflicts accessing the global stack have been avoided by assigning a private stack to each thread and that most of the markings a thread creates are processed by itself.

The time reduction for the CTMC generation by introducing local stacks can be seen in Figure 11 for Model A, which is also representative for the two other models. The improvement is substantial and — as expected — is more effective for post–elimination.

Figure 12 shows the dependency between the speedup of the ERG generation and the B–tree order σ for Model B applying post–elimination. A smaller choice for σ yields deeper B–trees, causing more locking operations, each of which, however, is of shorter duration. Increasing σ reduces the number of locking operations, but increases both their duration and the overall search cost. The best performance is obtained with $2 \leq \sigma_{opt} \leq 4$, independently of the number of processors. Elimination on–the–fly is not very sensitive to the B–tree order. Its performance decreases when orders higher than 20 are used.
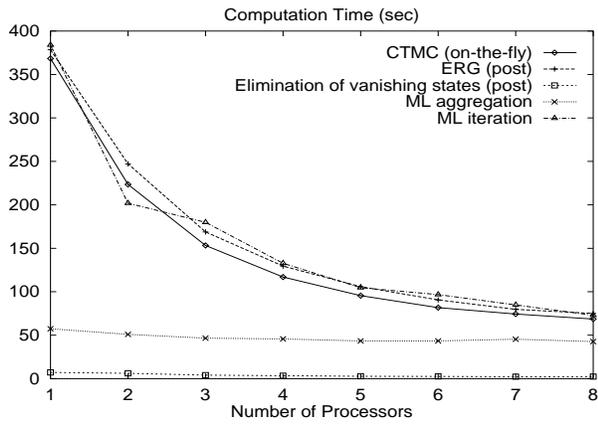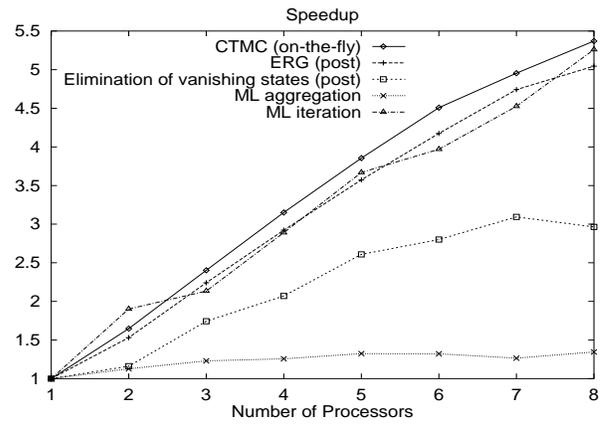
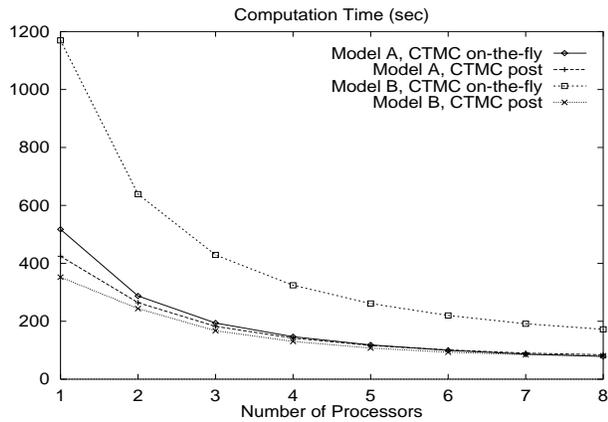Figure 6: Computation Times for Model C.

Figure 7: Speedups for Model C.

Figure 8: Computation Times of CTMC Generation, Models A and B with Post and On–the–fly Elimination.
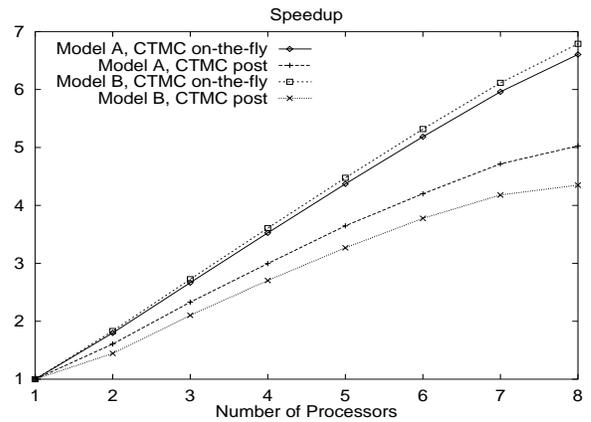

Figure 9: Speedups of CTMC Generation, Models A and B with Post and On–the–fly Elimination.
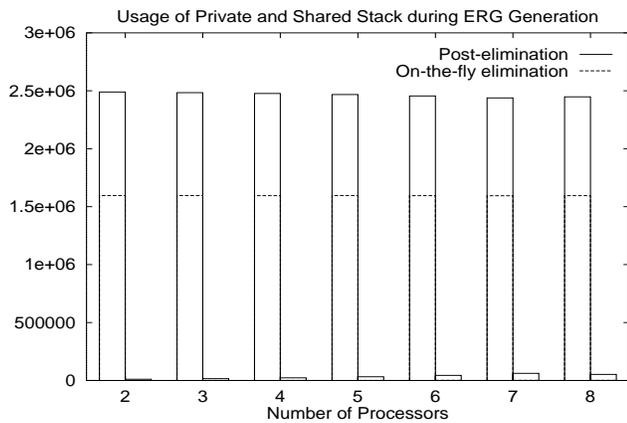

Figure 10: Private and Shared Stack Usage of CTMC Generation, Model A.
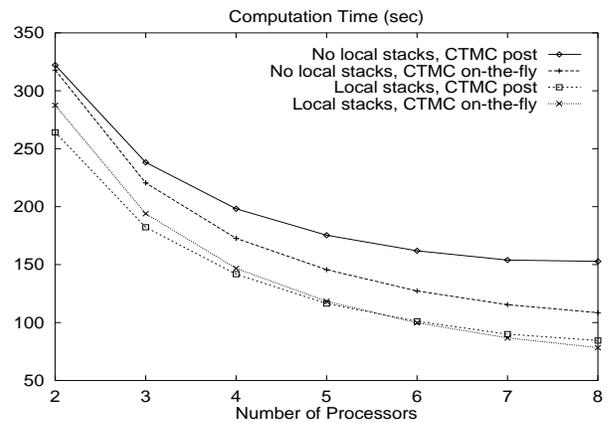

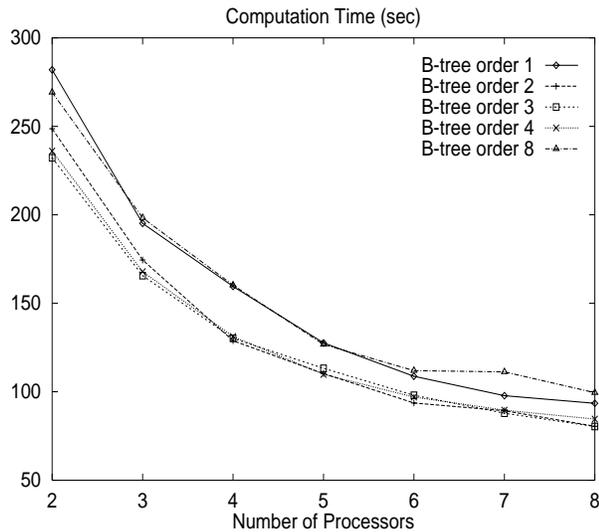Figure 11: Computation Times for CTMC Generation with and without Local Stacks, Model A.

Figure 12: Computation Time of ERG Generation for Various Values of σ, Model B with Post–elimination.
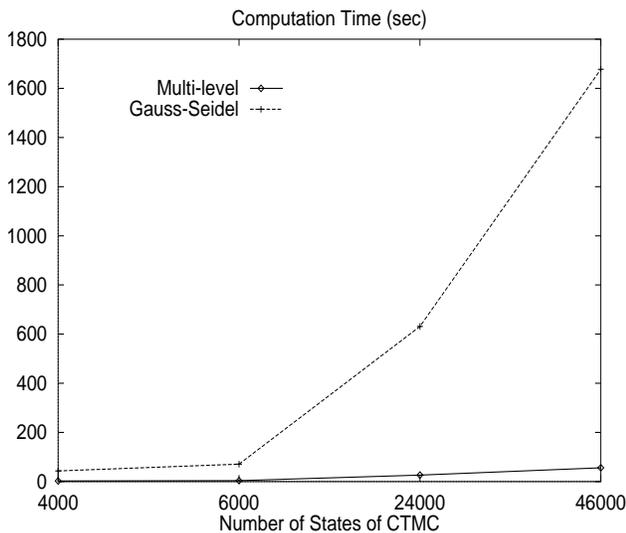

Figure 13: Computation Times for Gauss–Seidel and Multi–level for Different Sizes of Model B.

Figure 13 compares the computation times of the multi–level and the Gauss–Seidel methods using 8 processors for Model B with a stiff CTMC with up to about 46,000 states. The effect of the superior convergence rate of the multi–level algorithm can be clearly seen in the time difference, especially when bearing in mind that one multi–level iteration is about 10 times as expensive as one Gauss–Seidel iteration. In addition, the latter method can be more efficiently parallelized; the speedups obtained by Gauss–Seidel are approximately 15%–20% higher than those of the multi–level algorithm.

5 Conclusion and Future Work

We have described our parallelization concepts for both the state space construction of GSPNs and the numerical solution of the underlying CTMCs using a shared–memory multiprocessor. We considered an implementation of our algorithms and observed that our expectations were entirely met: the high performance and large main memory of a modern parallel computer could be efficiently exploited, allowing us to analyze large GSPN models that had been intractable using standard sequential methods on a single workstation. Furthermore, the performance gains of our parallel algorithms were nearly independent of the GSPN model, thus underlining the general usefulness of our concepts.

In contrast to contemporary distributed memory algorithms for parallel state space generation, no input from the modeler is required and no heuristics need to be applied for partitioning the state space in advance. Our approach to parallel state space generation mainly depends on the application of an efficient searching algorithm which is based on B–trees. These data structures support parallelism by providing efficient synchronization concepts.

The steady–state solution of the CTMCs is done using a parallel version of the iterative multi–level algorithm. This method has already proven to outperform the standard solution methods like Gauss–Seidel by achieving significantly higher rates of convergence, especially when applied to large, stiff problems.

Encouraged by the positive results of our work up to now, we intend to fine–tune our algorithms and implementations. Detailed measurements of the conflicts when concurrently accessing the search structure in the course of the reachability graph generation are to be done. Organizing a B–tree node as a balanced tree or using different node sizes for different B–tree levels might contribute to a better speedup. In the context of the multi–level method, we now have to focus on parallel aggregation strategies, because with the significant reduction of the time spent on the iterative computation, the aggregation phase has become the most time–consuming part of the solution process. In order that the parallel program be a useful modeling tool, we will add components for the evaluation of measures and rewards. These do not pose any particular difficulty with respect to parallelization.

We are at present developing parallel algorithms for distributed memory architectures and workstation clusters. Here, a different approach is necessary, and new problems such as efficient partitioning of the state space onto the processors and detecting the occurrence of load imbalance have to be solved. Our current ideas comprise the assignment of one subtree of the search structure to each processor and the application of automatic load balancing algorithms by observing the frequency of root splitting events.

References

[1] G. Balbo. On the success of stochastic Petri nets. In *Proc. IEEE Int. Workshop on Petri Nets and Performance Models (PNPM '95)*, pages 2–9, Durham, NC, 1995. IEEE Comp. Soc. Press.

[2] 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 (PNPM '87)*, pages 136–145, Madison, WI, USA, 1987.

[3] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9:1–21, 1977.

[4] P. Buchholz. Hierarchical high level Petri nets for complex system analysis. In *Proc. 15th International Conference on Applications and Theory of Petri Nets*, pages 24–31. Springer Verlag, Zaragoza, Spain, 1993. LNCS 185.

[5] S. Caselli, G. Conte, and P. Marenzoni. Parallel state space exploration for GSPN models. In *Proc. IEEE 16th Int. Conf. on Application and Theory of Petri nets*, Torino, Italy, 1995.

[6] G. Ciardo, A. Blakemore, P.F. Chimento, J.K. Muppala, and K.S. Trivedi. Automated generation and analysis of Markov reward models using stochastic reward nets. In C. Meyer and R.J. Plemmons, editors, *Linear Algebra, Markov Chains and Queueing Models*. Springer Verlag, 1992. IMA Volumes in Mathematics and its Applications, vol. 48.

[7] G. Ciardo, J. Gluckman, and D. Nicol. Distributed state–space generation of discrete–state stochastic models. *ORSA J. Comp.* To appear.

[8] G. Ciardo, J. Muppala, and K.S. Trivedi. SPNP: stochastic Petri net package. In *Proc. IEEE 3rd Int. Workshop Petri Nets and Performance Models (PNPM '89)*, pages 142–151, Kyoto, Japan, 1989.

[9] G. Ciardo, J.K. Muppala, and K.S. Trivedi. On the solution of GSPN reward models. *Performance Evaluation*, 12:237–253, 1991.

[10] D. Comer. The ubiquitous B-tree. *Computing Surveys*, 11(2):121–137, 1979.

[11] S. Greiner, A. Puliafito, G. Bolch, and K.S. Trivedi. Performance evaluation of dynamic priority operating systems. In *Proc. IEEE Int. Workshop on Petri Nets and Performance Models (PNPM '95)*, pages 241–250, Durham, NC, 1995. IEEE Comp. Soc. Press.

[12] L.J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th Symp. Foundations of Computer Science*, pages 8–21, 1978.

[13] G. Horton. A parallel multi–level solution method for large Markov chains. In David E. Keyes, Youcef Saad, and Donald G. Truhlar, editors, *Domain-Based Parallelism and Problem–Decomposition Methods in Computational Science and Engineering*. SIAM, Philadelphia, Pennsylvania, 1995.

[14] G. Horton and S. Greiner. Analysis of phase–type queueing networks with the multi–level method. In *Modelling and Simulation ESM '96*, Budapest, Hungary, 1996.

[15] G. Horton and S. Leutenegger. A multi–level solution algorithm for steady–state Markov chains. In *Proceedings of the ACM SIGMETRICS 1994 Conference on Measurement and Modeling of Computer Systems*, Nashville, Tennessee, 1994.

[16] M.A. Marsan, G. Balbo, A. Bobbio, G. Chiola, and G. Conte. The effect of execution policies on the semantics and analysis of stochastic Petri nets. *IEEE Trans. Software Engineering*, 15(7):832–846, 1989.

[17] M.A. Marsan, G. Balbo, G. Chiola, and S. Donatelli. On the product–form solution of a class of multiple-bus multiprocessor system models. *J. Systems and Software*, 1(2):117–124, 1986.

[18] M.A. Marsan, G. Balbo, and G. Conte. *Performance models of multiprocessor systems*. MIT Press, 1986.

[19] M.A. Marsan, S. Donatelli, F. Neri, and U. Rubino. On the construction of abstract GSPNs: An exercise in modeling. In *Proc. IEEE Int. Workshop Petri Nets and Performance Models (PNPM '91)*, Melbourne, Australia, 1991.

[20] R.A. Sahner and K.S. Trivedi. Reliability modeling using SHARPE. *IEEE Trans. Reliability*, R–36(2):186–193, 1987.

[21] W.H. Sanders and J.F. Meyer. Variable driven construction methods for stochastic activity networks. *Applied Mathematics and Performance/Reliability Models of Computer/Communication Systems*, 1987.

[22] W.H. Sanders and J.F. Meyer. Reduced Base Model Construction Methods for Stochastic Activity Networks. *IEEE J. Sel. Areas in Comm.*, 9(1), 1991.

[23] M. Sereno and G. Balbo. Computational algorithms for product form solution of stochastic Petri nets. In *Proc. IEEE Int. Workshop Petri Nets and Performance Models (PNPM '93)*, Toulouse, France, 1993. IEEE Comp. Soc. Press.

[24] W.J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, Princeton, New Jersey, 1994.

[25] K.S. Trivedi and M. Malhotra. Reliability and performability techniques and tools: A survey. In B. Walke and O. Spaniol, editors, *Messung, Modellierung und Bewertung von Rechen– und Kommunikationssystemen*, pages 27–48. Springer, 1993.

[26] P. Ziegler. A structure based decomposition approach for GSPN. In *Proc. IEEE Int. Workshop on Petri Nets and Performance Models (PNPM '95)*, pages 261–270, Durham, NC, 1995. IEEE Comp. Soc. Press.