Parallel Shared–Memory State–Space Exploration in Stochastic Modeling

Susann C. Allmaier, Graham Horton

Computer Science Department III, University of Erlangen–Nürnberg, Martensstr. 3, 91058 Erlangen, Germany. Email : {*snallmai* | *graham*}*@informatik.uni-erlangen.de.*

Abstract. Stochastic modeling forms the basis for analysis in many areas, including biological and economic systems, as well as the performance and reliability modeling of computers and communication networks. One common approach is the state–space–based technique, which, starting from a high–level model, uses depth–first search to generate both a description of every possible state of the model and the dynamics of the transitions between them. However, these state spaces, besides being very irregular in structure, are subject to a combinatorial explosion, and can thus become extremely large. In the interest therefore of utilizing both the large memory capacity and the greater computational performance of modern multiprocessors, we are interested in implementing parallel algorithms for the generation and solution of these problems. In this paper we describe the techniques we use to generate the state space of a stochastic Petri–net model using shared–memory multiprocessors. We describe some of the problems encountered and our solutions, in particular the use of modified B–trees as a data structure for the parallel search process. We present results obtained from experiments on two different shared–memory machines.

1 Introduction

Stochastic modeling is an important technique for the performance and reliability analysis of computer and communication systems. By performing an analysis of an appropriate abstract model, useful information can be gained about the behavior of the system under consideration. Particularly for the validation of a system concept at an early design stage, values for expected performance and reliability can be obtained. Typical quantities of interest in computer performance might be the average job throughput of a server or the probability of buffer overflow of a network node. In reliability analysis, probabilities for critical system states such as failures may be computed. As a result of such analyses, design parameters such as protocol algorithms, degrees of redundancy and component bandwidths may be optimized. Thus the ability to perform these analyses quickly and efficiently is of great importance [2].

One important class of techniques for stochastic modeling beside analytical and discrete–event simulation approaches is state–space analysis. Here, a high–level model such as a queuing network or stochastic Petri net is created, from which the entire state–space graph is generated, in which there is one node for

each possible state which the model can assume. The states are linked by arcs which describe the timing characteristics for each state change. The state space is thus described by an annotated directed graph.

Using the simplest and most common assumptions on the transitions — that the time spent by the system in each state is exponentially distributed — the stochastic process described by the model is a Markov chain. In this case, the directed graph of the state space represents a matrix and the transient and steady–state analysis is performed by solving a corresponding system of ordinary differential equations and linear system of equations respectively.

Owing to the combinatorial nature of the problem, the state spaces arising in practical problems can be extremely large ($> 10^6$ nodes and arcs). The memory and computing requirements for the resulting systems of equations grow correspondingly. It is the size of the state space that is the major limiting factor in the application of these modeling techniques for practical problems. This motivates the investigation of parallel computing for this type of stochastic analysis.

One well–known technique for describing complex stochastic systems in a compact way are Generalized Stochastic Petri Nets (GSPNs) [7, 8]. Petri nets allow a graphical and easily understandable representation of the behavior of a complex system, including timing and conditional behavior as well as the forking and synchronization of processes.

We are interested in using shared–memory multiprocessors for the analysis of GSPN models. Such machines are becoming more widespread both as mainframe supercomputers and also as high–end workstations and servers. The shared–memory programming model is more general than the message–passing paradigm, allowing, for example, the concurrent access to shared data structures. On the other hand, the model contains other difficulties such as contention for this access, which requires advanced synchronization and locking methods. These will be the subject of this paper. We consider implementations on two different shared memory multiprocessors: the Convex Exemplar SPP1600 mainframe supercomputer using the proprietary CPS thread programming environment and a Sun Enterprise 4000 multiprocessor server using POSIX threads.

The significance of this work lies in the extension of the ability to model with GSPNs to shared–memory multiprocessors. To our knowledge, there has, until now, been no work published concerning parallel shared–memory state–space generation for stochastic models. An approach for distributed memory machines was published in [4]. The results of this work should provide faster state–space generation and, in conjunction with parallel numerical algorithms, overall acceleration of the analysis process. In particular, we will be able to better utilize the larger main memories of modern shared–memory multiprocessors. This will also enable the analysis of models whose size has prevented their computation on standard workstations.

In the following section we describe state–space generation for Petri nets. In Section 3 we describe the parallelization issues and the solution techniques we used. Section 4 contains results from the parallel programs and in Section 5 we give a short conclusion.

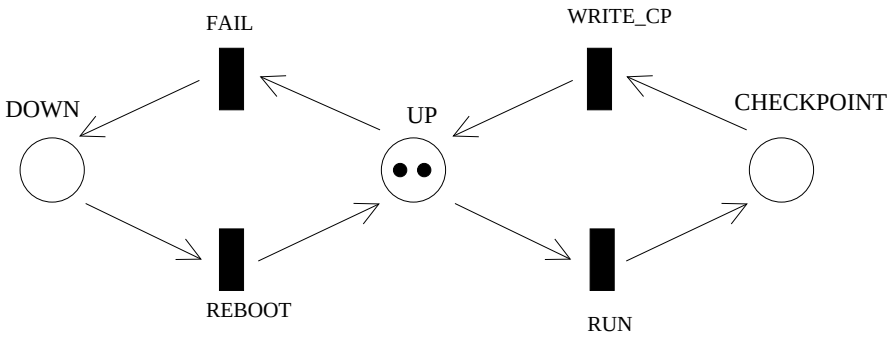

Fig. 1. Example GSPN Model of a Computer System with Failures and Checkpoints.

2 State–Space Generation for GSPNs

In this section we briefly describe stochastic Petri nets and the automatic generation of their underlying state spaces.

One of the most widely used high–level paradigms for stochastic modeling are *Generalized Stochastic Petri Nets (GSPNs)* [7, 8]. These are an extension to standard Petri nets to allow stochastic timed transitions between individual states. They have the advantages of being easy to understand and having a natural graphical representation, while at the same time possessing many useful modeling features, including sequence, fork, join and synchronization of processes. The state space, or *reachability graph*, of a GSPN is a semi–Markov process, from which states with zero time are eliminated to create a Markov chain. The Markov chain can be solved numerically, yielding probability values which can be combined to provide useful information about the net.

A GSPN is a directed bipartite graph with nodes called *places*, represented by circles, and *transitions*, represented by rectangles. Places may contain *tokens*, which are drawn as small black circles. In a GSPN, two types of transitions are defined, *immediate transitions*, and *timed transitions*. For simplicity, we will not consider immediate transitions any further in this paper. If there is a token in each place that is connected to a transition by an *input arc*, then the transition is said to be *enabled* and may *fire* after a certain delay, causing all these tokens to be destroyed, and creating one token in each place to which the transition is connected by an *output arc*. The state of the Petri net is described by its *marking*, an integer vector containing the number of tokens in each place. The first marking is commonly known as the *initial marking*. A timed transition that is enabled will fire after a random amount of time that is exponentially distributed with a certain rate.

Figure 1 shows a small example GSPN that models a group of computers, each of which may either be running (UP), writing a checkpoint (CHECKPOINT), or failed and rebooting from the last checkpoint (DOWN). The changes between these states are represented by timed transitions with appropriate exponentially distributed rates. In this case, we have modeled two computers (by

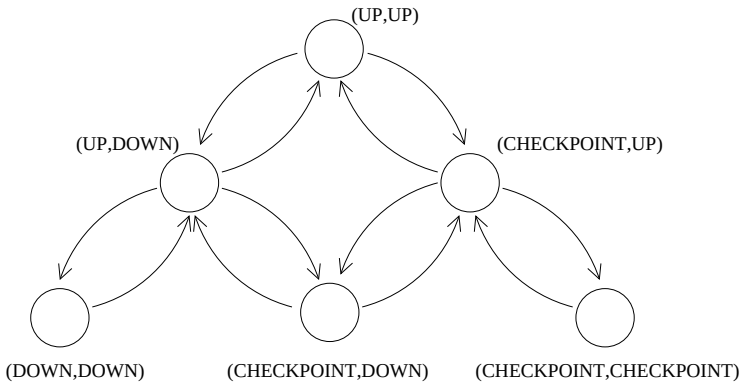

Fig. 2. State Space for Example GSPN.

inserting two tokens into the net, initially in place UP). Note that we could model any number of computers by simply adding tokens accordingly, assuming that the transition rates are independent of the states of the other machines.

Figure 2 shows the state space, or *reachability graph*, of this GSPN with two tokens. For readability, we have omitted the rates that are attached to the arcs and have used a textual description for the marking vector. Each state corresponds to one possible marking of the net, and each arc to the firing of a transition in that marking. Owing to the simplicity of this particular Petri net, the state space has a regular triangular structure. In general, however, the reachability graph is highly irregular.

In this example, the state–space graph has six nodes. It is easy to see that adding tokens to the net will lead to a rapid increase in the size of the graph. In the general case, the size grows as t^p, where t is the initial number of tokens and p the number of places in the Petri net. It is for this reason that Petri nets of even moderate complexity may have state spaces whose storage requirements exceed the capacities of all but the largest of computers. In addition, the computation times for the solution of the underlying equations grows accordingly. It is, of course, for these reasons that we are interested in parallel computation.

Figure 3 shows the sequential state–space generation algorithm in pseudo–code form. It utilizes a stack S and a data structure D, which is used to quickly determine whether or not a newly detected marking has previously been discovered. D is typically chosen to be either a hash table or a tree. One of the contributions of this work is the use of a modified B–tree to allow rapid search whilst at the same time minimizing access conflicts. The algorithm performs a depth–first search of the entire state space by popping a state from the stack, generating all possible successor states by firing each enabled transition in the Petri net and pushing each thus–created new marking back onto the stack, if it is one that has not already been generated. Replacing the stack by a FIFO memory would result in a breadth–first search strategy.

```
 1 procedure generate_state_space
 2 initial marking m_0
 3 reachability graph R = ∅
 4 search data structure D = ∅
 5 stack S = ∅
 6 begin
 7    add state m_0 to R
 8    insert m_0 into D
 9    push m_0 onto S
10    while (S ≠ ∅)
11          m_i = pop(S)
12          for each successor marking m_j to m_i
13              if (m_j ∉ D)
14                 add state m_j to R
15                 insert m_j into D
16                 push m_j onto S
17              endif
18              add arc m_i → m_j to R
19          endfor
20    endwhile
21 end generate_state_space
```

Fig. 3. Sequential State–Space Generation Algorithm

3 Parallelization Issues

The state–space generation algorithm is similar to other state–space enumeration algorithms, such as the branch–and–bound methods used for the solution of combinatorial problems such as computer chess and the traveling salesman problem. Consequently it presents similar difficulties in parallelization, namely

- The size of the state space is unknown *a priori*. For many Petri nets it cannot be estimated even to within an order of magnitude in advance.
- All processors must be able to quickly determine whether a newly generated state has already been found — possibly by another processor — to guarantee uniqueness of the states. This implies either a mapping function of states to processors or an efficiently implemented, globally accessible search data structure.
- The state space grows dynamically in an unpredictable fashion, making the problem of load balancing especially difficult.

However, there are also two significant differences to a branch–and–bound algorithm:

- The result of the algorithm 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.
- No cutoff is performed, i.e. the entire state space must be generated.

Our parallelization approach lets different parts of the reachability graph be generated simultaneously. This can be done by processing the main loop of algorithm `generate_state_space` (Lines 10–20) concurrently and independently on all threads, implying simultaneous read and write access to the global data structures R, D and S. Thus the main problem is maintaining data consistency on the three dynamically changing shared data structures in an efficient way. Our approach applies two different methods to solve this problem: S is partitioned onto the different threads employing a shared stack for load balancing reasons only, whereas the design of D and R limits accesses to a controlled part of the data structure which can be locked separately.

With respect to control flow there is not much to say: threads are spawned before entering the main loop of algorithm `generate_state_space` and termination is tested only rarely, namely when an empty shared stack is encountered[1]. Thus we can concentrate on the crucial and interesting part of the problem: the organization of the global data structures and the locking mechanisms that we have designed.

3.1 Synchronization

Synchronization is done by protecting portions of the global shared data with mutex variables providing mutual exclusive access.

Because arcs are linked to the data structures of their destination states (m_j in Figure 3), rather than their source states (m_i in Figure 3), synchronization for manipulating the reachability graph may be restricted to data structure D: marking m_j is locked implicitly when looking for it in D by locking the corresponding data in D and holding the lock until Line 18 has been processed. No barriers are needed in the course of the generation algorithm.

Synchronization within the Search Data Structure. We first considered designing the search data structure D as a hash table like some (sequential) GSPN tools do, because concurrent access to hash tables can be synchronized very easily, by locking each entry of the hash table before accessing it. But there are many unsolved problems in using hash tables in this context: As mentioned earlier, neither the size of the state space is known in advance — making it impossible to estimate the size of the hash table *a priori* — nor is its shape, which means that the hash function which maps search keys onto hash table entries would be totally heuristic.

For these reasons, we decided to use a balanced search tree for retrieving already generated states. This guarantees retrieval times that grow only logarithmically with the number of states generated.

The main synchronization problem in search trees is rebalancing: a non–balanced tree could be traversed by the threads concurrently by just locking the tree node they encounter, unlocking it when progressing to the next one, and,

[1] This can easily be done by setting and testing termination flags associated with threads under mutual exclusion.

if the search is unsuccessful, inserting a new state as a leaf without touching the upper part of the tree. Rebalancing — obviously obligatory for efficiency reasons with large state spaces — means that inserting a new state causes a global change to the tree structure. To allow concurrency, the portion of the tree that can be affected by the rebalance procedure must be anticipated and restricted to as small an area as possible, since this part has to be locked until rebalancing is complete. A state is looked up in D for each arc of R (Line 13 in Figure 3), which will generate a lot of contention if no special precautions are taken.

We found an efficient way to maintain the balance of the tree by allowing concurrent access through the use of B–trees ([5]): these are by definition automatically balanced, whereby the part of the tree that is affected by rebalancing is restricted in a suitable way.

Synchronization Schemes on B–trees. A B–tree node may contain more than one search key — which is a GSPN marking in this context. The 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.

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. New keys are always inserted 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.

The entity that can be locked is the B–tree node containing several keys. Several methods which require one or more mutex variables per node are known [3]. We have observed that using more than one mutex variable causes an unjustifiable overhead, since the operations on each state consume little time. The easiest way to avoid data inconsistencies is to lock each node encountered on the way down the tree during 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 even do not know a priori if an insertion will actually take place, we have developed another method adapted from [6] 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. This way back propagation does not occur, since parent nodes can never be full. Therefore a thread holds at most one lock at

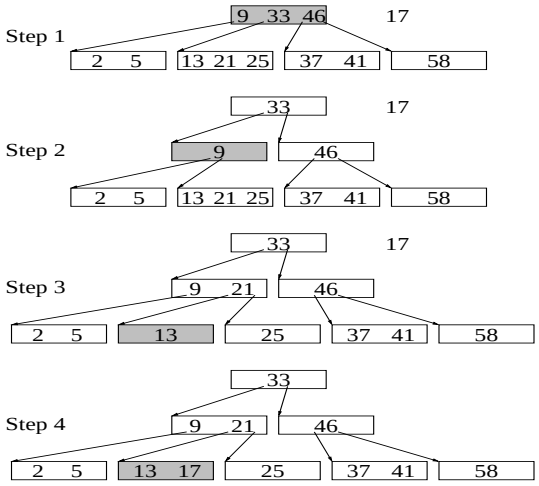

Fig. 4. Splitting–in–advance when Inserting Key 17 in a B–tree

a time. The lock moves down the tree as the search proceeds. For this reason, access conflicts between threads are kept to a small number, thus allowing high concurrency of the search tree processing. Figure 4 shows the insertion of key 17 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 efficient storage methods, the organization of the data is similar to that of binary trees, whereby B–tree states consume at most one more byte per state than binary trees [5].

Synchronization on the Stack. The shared stack, which stores the as yet unprocessed markings, is the pool from which the threads get their work. In this sense the shared stack implicitly does the load balancing and therefore cannot be omitted. Since it has to be accessed in mutual exclusion, one shared stack would be a considerable bottleneck, forcing every new marking to be deposited there regardless if all the threads are provided with work anyway.

Therefore we additionally assign a private stack to each thread. Each thread uses mainly its private stack, only pushing a new marking onto the shared stack if the latter's depth drops below the number of threads N. A thread only pops markings from the shared stack if its private stack is empty. In this manner load imbalance is avoided: a thread whose private stack has run empty because it has generated no new successor markings locally, is likely to find a marking on the shared stack, provided the termination criterion has not been reached.

The shared stack has to be protected by a mutex variable. The variable containing the stack depth may be read and written with atomic operations,

thus avoiding any locking when reading it. This is due to some considerations:

- The variable representing the depth of the shared stack can be stored in one byte because its value is always smaller than two times the number of threads: if the stack depth is $n-1$ when all the threads are reading it every thread will push a marking there leading to a stack depth of $2N-1 > N$ which causes the threads to use their private stacks again.
- The number of threads can be restricted to $N = 256 = 2^8$ so that $2N-1$ can be represented in one byte without any loss of generality since the state–space generation of GSPNs is no massively parallel application.

3.2 Implementation Issues

Synchronization. Since we have to assign a mutex variable to each B–tree node in the growing search structure, our algorithm relies on the number of mutex variables being only limited by memory size.

Our algorithm can be adapted to the overhead that lock and unlock operations cause on a given machine by increasing or decreasing the order of the B–tree σ: an increase in σ saves mutex variables and locking operations. On the other hand a bigger σ increases both overall search time — since each B–tree node is organized as linear list — and search time within one node, which also increases the time one node stays locked. Measurements in Section 4 will show that the savings in the number of locking operations are limited and that small values for σ lead to a better performance for this reason.

Waiting at Mutex Variables. Measurements showed that for our algorithm — which locks small portions of data very frequently — it is very important that the threads perform a spin wait for locked mutex variables rather than getting idle and descheduled. The latter method may lead to time consuming system calls and voluntary context switches in some implementations of the thread libraries where threads can be regarded as light weight processes which are visual to the operating system (e.g. Sun Solaris POSIX threads). Unfortunately busy waits make a tool–based analysis of the waiting time for mutex variables difficult since idle time is converted to CPU time and gets transparent to the analysis process.

Memory Management. Our first implementation of the parallel state–space generation algorithm did not gain any speedups at all. This was due to the fact that dynamic memory allocation via the `malloc()` library function is implemented as a mutually exclusive procedure. Therefore two threads that each allocate one item of data supposedly in parallel always need more time that one thread would need to allocate both items.

Since our sparse storage techniques exploit dynamic allocation intensively, we had to implement our own memory management on top of the library functions: each thread reserves large chunks of private memory and uses special `allocate()` and `free()` functions for individual objects. In this way, only few memory allocation must be done under mutual exclusion.

4 Experimental Results

We implemented our algorithms using two different shared–memory multiprocessors:

- A Convex Exemplar SPP1600 multiprocessor with 4 Gbytes of main memory and 32 Hewlett/Packard HP PA-RISC 7200 processors. It is a UMA (uniform memory access) machine within each hypernode subcomplex consisting of 8 processors whereby memory is accessed via a crossbar switch. Larger configurations are of NUMA (non–uniform memory access) type, as memory accesses to other hypernodes go via the so–called CTI ring. We did our measurements on a configuration with one hypernode to be able to better compare with the second machine:
- A Sun Enterprise server 4000 with 2 Gbytes of main memory and 8 UltraSparc–1 processors which can be regarded as UMA since memory is always accessed via a crossbar switch and a bus system.

Our experiments use a representative GSPN adapted from [7]. It models a multiprocessor system with failures and repairs. The size of its state space can be scaled by initializing the GSPN with more or less tokens that represent the processors of the system. The state spaces we generated consist of 260,000 states and 2,300,000 arcs (Size S) and of 900,000 states and 3,200,000 arcs respectively (Size M).

Figure 5 shows the overall computation times needed for the reachability graph generation depending on the number of processors measured on the Convex SPP and on the Sun Enterprise for the GSPN of Size M. Figure 6 shows the corresponding speedup values and additionally the speedups for the smaller model (Size S). In the monoprocessor versions used for these measurements, all parallelization overhead was deleted and the B–tree order was optimized to $\sigma = 1$. The figures show the efficiency of our algorithms — especially of the applied synchronization strategies. For both architectures the speedup is linear. [1] shows that these speedups are nearly model independent.

Figure 7 shows the dependency between the computation times of the state–space generation and the B–tree order σ for Size M measured on the Sun Enterprise.

Table 1 gives the total number of locking operations and the number of used mutex variables for different B–tree orders σ for a parallel run with 8 processors. It can be seen that the number of locking operations decreases only by a factor of 3.3 whereby the total number of mutex variables — which is also the number of B–tree nodes — decreases by a factor of 34.7 when σ is increased from 1 to 32. This is due to the fact that for each arc in the state space at least one locking operation has to be performed and that the number of locking operations per arc depends only on how deep the search moves down the B–tree (see Section 3.1). Thus it becomes intelligible that $\sigma < 8$ leads to the best performance (compare Section 3.2).

The number of shared stack accesses turned out to be negligible when local stacks are used: 50 was the maximum number of markings ever popped from

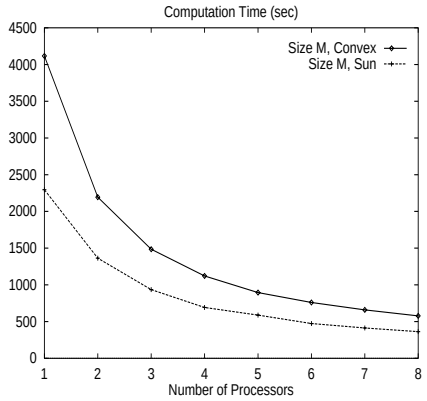

Fig. 5. Computation Times, Convex and Sun, Size M.

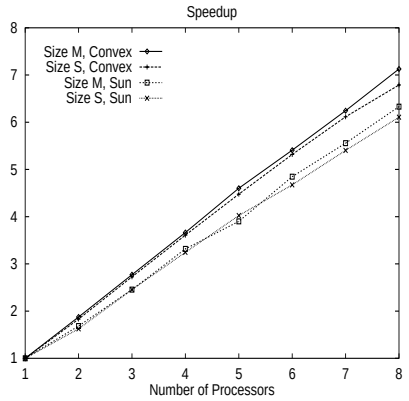

Fig. 6. Speedups, Convex and Sun, Size M and S.

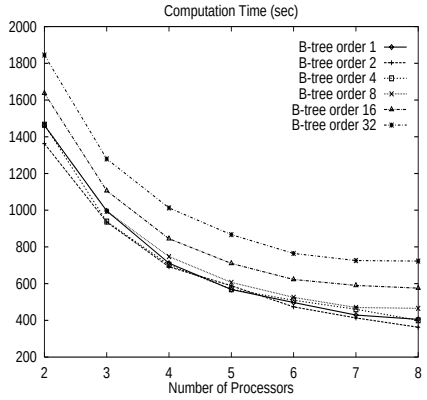

Fig. 7. Computation Times for Various Values of σ, Sun, Size M.

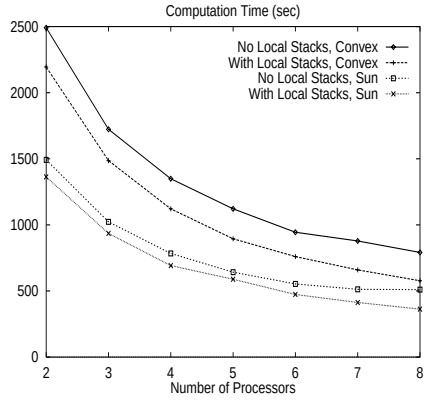

Fig. 8. Computation Times with and without Local Stacks, Convex and Sun, Size M.

B–tree order σ	Number of Locking Operations	Number of Mutex Variables
1	165,000,000	520,000
2	110,000,000	255,000
4	75,000,000	125,000
8	65,000,000	60,000
16	55,000,000	30,000
32	50,000,000	15,000

Table 1. Synchronization Statistics for Size M

the shared stack during all our experimental runs. Figure 8 compares the computation times with and without the use of local stacks on both multiprocessor architectures for Size M.

5 Conclusion and Further Work

We presented a parallel algorithm for generating the state space of GSPNs which is mainly based on the use of modified B–trees as a parallel search data structure. Measurements showed good linear speedups on different architectures.

One B–tree node could be organized as a balanced tree rather than a linear list. But measurements of the reduction in the number of locking operations when σ is increased (Table 1) let expect only moderate performance improvements this way.

On the other hand the maintenance of several B–trees rather than one seems to be a a promising improvement in the organization of the search data structure: conflicts at the root node could be avoided thus allowing a higher degree of parallelization.

Acknowledgments. We wish to thank Stefan Dalibor and Stefan Turowski at the University of Erlangen–Nürnberg for their helpful suggestions and for their assistance in the experimental work.

References

1. 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 (PNPM '97)*, St. Malo, France, 1997. IEEE Comp. Soc. Press. To appear.
2. 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.
3. R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9:1–21, 1977.
4. 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.
5. D. Comer. The ubiquitous B-tree. *Computing Surveys*, 11(2):121–137, 1979.
6. L.J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th Symp. Foundations of Computer Science*, pages 8–21, 1978.
7. M. Ajmone Marsan, G. Balbo, and G. Conte. *Performance models of multiprocessor systems.* MIT Press, 1986.
8. M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with generalized stochastic Petri nets.* Wiley, Series in Parallel Computing, 1995.