Stochastic Graph Models for Performance Evaluation of Parallel Programs and the Evaluation Tool *PEPP*

Franz Hartleb
University of Erlangen–Nürnberg, IMMD 7,
Martensstraße 3, 91058 Erlangen, Germany

Abstract

For parallelizing an algorithm and for mapping a given program onto a parallel or distributed system there are generally many possibilities. Performance models can help to predict which implementation and which mapping is the best for a given algorithm and for a given computer configuration. Stochastic graph modeling is an appropriate method, since the execution order of tasks, their runtime distribution, and branching probabilities are represented.

In this paper a survey of the modeling possibilities and the analysis techniques implemented in our tool PEPP is presented. The analysis techniques include the series-parallel reduction applied on the numerical representation of the tasks's runtime, a new approximation method, and powerful bounding methods for the mean runtime.

1. Introduction

In order to minimize the time for solving big problems, given problems are divided in subproblems which are then solved in parallel with $p > 1$ processors. During the search for optimal parallel solutions the execution times of different algorithms and implementation strategies have to be evaluated. Since implementing these different versions is too costly or even impossible if the desired hardware is not available, predicting the runtime with a model is an appropriate method.

As the runtime of a user program normally depends on the input values and on the mostly unknown duration of library and system functions it is not easy to predict it, even if a program is running on a monoprocessor system. In parallel systems where processes communicate with each other and share resources there are additional factors that have an influence on the runtime, e.g. race conditions, mutual waiting of processes, or access conflicts on interconnection networks. Therefore, modeling the behavior of parallel programs in order to predict their runtime can be very complicated.

Graph models have been used extensively in order to model and to analyze the behavior of parallel programs [20, 19]. Experiences showed that realistic models often consist of

hundreds of nodes. If large models are not series-parallel structured, the exact evaluation methods fail because of too high computation times. Two ways of overcoming this problem are approximate methods or methods to obtain lower and upper bounds of the mean runtime. Both methods which need much less computation time. We have developed methods and tools for both ways.

If one knows the runtimes of the essential parts of the parallel program it is possible to use this knowledge for predicting the performance of not yet implemented alternatives. This is done for analyzing open problems, such as different synchronization methods or interconnection networks, based on numerical distribution functions. Additionally, the integration of measurement results and models for performance prediction allows us to increase the accuracy of the predicted runtime.

Additionally, the analysis based on numerical functions instead of parametric functions has two essential advantages. First it is much more efficient and second modeling the tasks' runtime is not restricted to a particular type of distribution functions like exponential or phase type distribution functions.

This paper is part of an effort to develop methods and a tool PEPP (Performance Evaluation of Parallel Programs) [2] for evaluating the performance of parallel programs. All tools are based on graph models. PEPP is used for automatic instrumentation of C-programs [8] and to predict the runtime distribution, the mean runtime, or bounds for the mean runtime of parallel programs. In section 2 we define the stochastic graph model, the node types, and the runtime distributions used in PEPP. A survey of the analysis techniques developed is given in section 3. The accuracy of the approximation technique and the quality of the bounding methods are shown in section 4.

2. Stochastic Graph Models

A parallel program is modeled by an acyclic, directed, stochastic graph $G = (V, E, \underline{T})$ which consists of a set of nodes V, a set of directed edges $E \subset V \times V$ (arcs), and a vector of random variables $\underline{T}$. Each task or module t_i of the parallel program is modeled by one node v_i and the corresponding random variable T_i which describes the runtime behavior of task t_i. The tasks' runtimes are assumed to be independent random variables. The dependency between tasks is modeled by arcs. An arc from node v_i to node v_j means that task t_j can start execution only if the execution of task t_i is finished. In the following section, we also use the set of predecessor nodes $p(v_i)$ and the set of successor nodes $s(v_i)$ of node v_i. If $p(v_i) = \emptyset$, then v_i is called a start node. v_i is called an end node, if $s(v_i) = \emptyset$. The level $l(v_i)$ of node v_i is defined as the longest path from a start node to node v_i:

$$l(v_i) = \begin{cases} 0 & \text{if } p(v_i) = \emptyset \\ \max\limits_{v_j \in p(v_i)} l(v_j) + 1 & \text{else} \end{cases} \quad (1)$$

2.1. Node Types in PEPP

Obviously, there is a gap between programs which are not acyclic since they contain loops and acyclic graph models. In order to bridge over this gap and to make modeling and model evaluation more efficient in PEPP, we define four node types:

- An *elementary node* represents one task t_i of the parallel program with service time distribution $F_i(t)$.
- The execution of n identical tasks on n processors in parallel is modeled by a *parallel node*. All the n tasks are assumed to have the same runtime distribution $F_i(t)$.
- Iterations and loops are modeled by *cyclic nodes* or *hierarchical loop nodes*.
  A cyclic node is characterized by the runtime distribution of the body of the iteration or loop and the probabilities p_k which are the probabilities that the body is executed again after the k-th iteration. The body of a hierarchical loop node may be described by any PEPP graph. Both node types allow us to model cycles and iterations and the graph remains acyclic.
- A *hierarchical node* contains an arbitrary graph model consisting of elementary, parallel, cyclic, and hierarchical nodes. Any graph model can be included in a higher order graph by this node type.

2.2. Runtime Distributions

For the tasks' runtime distribution $F(t)$ many parametric distributions and numerical distributions can be used in PEPP. The runtime distribution may be

- deterministic $F(t) = \begin{cases} 0 & \text{if } t < d \\ 1 & \text{else} \end{cases}$
- exponential $F(t) = 1 - e^{-\lambda t}$
- Branching Erlang $F(t) = 1 - \sum_{n=1}^{N} p_n e^{-\lambda_n t} \sum_{j=0}^{k_n - 1} \frac{(\lambda_n t)^j}{j!}$
- one deterministic and one exponential phase $F(t) = \begin{cases} 1 - e^{-\lambda(t-d)} & \text{if } t \geq d \\ 0 & \text{else} \end{cases}$
- numerically given

The deterministic, the exponential, and the Branching Erlang distribution are well known and often used in performance models. In many cases the approximation of the real runtime behavior with deterministic or exponential distributions is not appropriate and the approximation with Branching Erlang distributions often leads to a very high number of phases which makes the model intractable. In order to avoid the problems of inappropriateness and intractability we developed two approaches for describing runtime distributions.

Approximating the runtime with one deterministically and one exponentially distributed phase (de-approximation) allows us to adapt the first and the second moment of the runtime and we get only two phases for one node.

In the second approach we describe any parametric or measured distribution function by its numerical representation (figure 1). A numerical density function f of an arbitrary

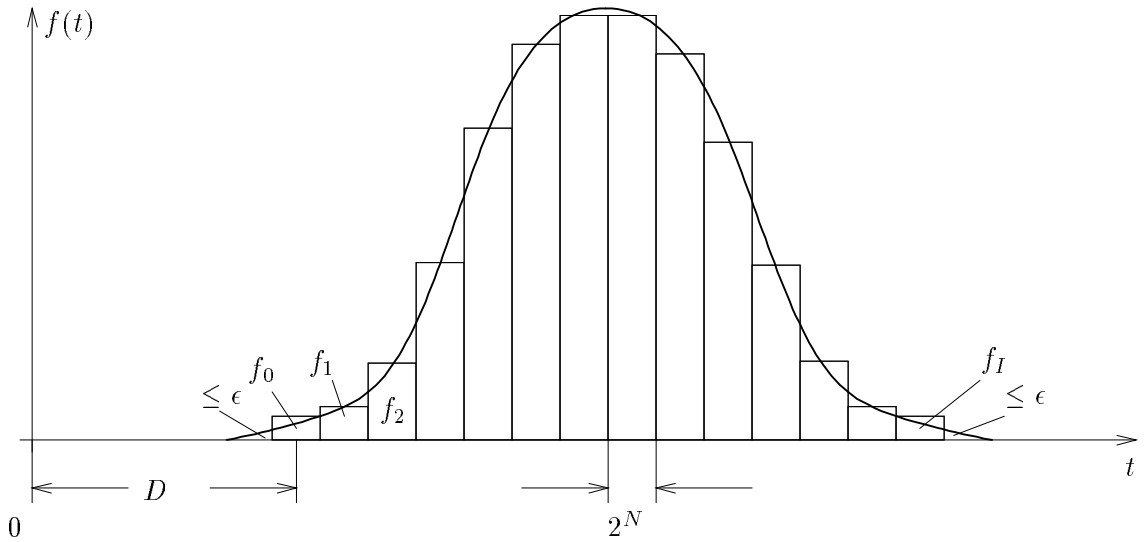

Figure 1: Continuous density function and the corresponding numerical function

density function $f(t)$ is defined as a tupel $f = (D, N, f_0, f_1, \ldots, f_I)$ [9] with displacement D, order N, and

$$f_0 = \int_0^{D+2^{N-1}} f(t)\,dt; \qquad f_i = \int_{D+(2i-1)2^{N-1}}^{D+(2i+1)2^{N-1}} f(t)\,dt; \qquad f_I = \int_{D+(2I-1)2^{N-1}}^{\infty} f(t)\,dt \tag{2}$$

The width of the discrete steps is defined by the order N. To make the order of different densities adaptable, we always choose $D \bmod 2^N = 0$. This is important for model analysis. The numerical density is called ϵ–exact, if

$$\int_0^{D-2^{N-1}} f(t)\,dt \leq \epsilon \qquad \wedge \int_{D+(2I+1)2^{N-1}}^{\infty} f(t)\,dt \leq \epsilon \tag{3}$$

This definition keeps the number of intervals small by neglecting the left and right tail of the density function. In our implementation ϵ is 10^{-6}.

3. Model Analysis

3.1. Series-parallel Reduction

The analysis method depends heavily on the structure of a given graph. In order to classify the models in *series-parallel* and *non-series-parallel* graphs we define two reduction operators:

- *serial reduction*: This operator can be applied on two nodes v_i and v_j, if $s(v_i) = \{v_j\}$ and $p(v_j) = \{v_i\}$. By the serial reduction operator the two nodes v_i and v_j are replaced in the graph model by one node v_{sr} with $p(v_{sr}) = p(v_i)$, $s(v_{sr}) = s(v_j)$, and

$$F_{sr}(t) = \int_0^t F_i(\tau) f_j(t-\tau)\,d\tau \qquad \text{(convolution)} \tag{4}$$

If the tasks' runtime distributions are given by numerical descriptions f_i and f_j, f_{sr} is obtained from

$$\begin{aligned} D_{sr} &= D_i + D_j \\ N_{sr} &= N_i = N_j \\ I_{sr} &= I_i + I_j \qquad\qquad \text{(num. convolution)} \\ f_k^{sr} &= \sum_{l=\max(0,k-I_i)}^{\min(I_i,k)} f_l^i f_{k-l}^j \end{aligned} \tag{5}$$

- *parallel reduction*: Two nodes v_i and v_j can be reduced to one node v_{pr}, if $p(v_i) = p(v_j)$ and $s(v_i) = s(v_j)$. The two nodes are replaced by one node v_{pr} with $p(v_{pr}) = p(v_i)$, $s(v_{pr}) = s(v_i)$, and $F_{pr}(t) = F_i(t)F_j(t)$.

In the numerical case, the product is built by

$$\begin{aligned} D_{pr} &= \max(D_i, D_j) \\ N_{pr} &= N_i = N_j \\ I_{pr} &= \max(I_i - d_i, I_j - d_j) \text{ with } d_i = \frac{D_j - D_i}{2^{N_i}}; d_j = \frac{D_i - D_j}{2^{N_j}}; \\ f_k^{pr} &= f_{k+d_i}^i \left(\frac{1}{2} f_{k+d_j}^j + \sum_{l=0}^{k-1+d_j} f_l^j \right) + f_{k+d_j}^j \left(\frac{1}{2} f_{k+d_i}^i + \sum_{l=0}^{k-1+d_i} f_l^i \right) \end{aligned} \tag{6}$$

A graph G is called a series-parallel graph if G can be reduced to one node by the operators serial reduction and parallel reduction. All other graphs are called non-series-parallel graphs.

Obviously, if we have a series-parallel graph, we are able to obtain the runtime distribution of the modeled parallel program by reducing the graph to one node v_{sp}. The random variable T_{sp} corresponding to v_{sp} describes the runtime behavior, and the mean runtime as well as higher moments can be obtained. This technique was used in [14, 16, 15] and [5] to analyze series-parallel graphs where the tasks' runtime distributions are modeled by exponential polynomials. However, applying this method the number of parameters grows so fast that it can be used only for small graphs. We cope with this problem by transforming the parametric functions to numerical functions and applying the reduction operators to the numerical functions [9, 13]. This allows us to evaluate series-parallel graphs of nearly arbitrary size.

If a program can be modeled only by a non-series-parallel graph, PEPP offers two evaluation methods, an approximate state space analysis and bounding methods.

3.2. Transient State Space Analysis

The exact analysis of non-series-parallel graphs is very costly, even for small graphs. In [10] a method is introduced to compute the runtime distribution of non-series-parallel structured graphs in which the tasks' runtimes are described by polynomial functions. If the tasks' runtime distributions are given by General Erlang functions, we can use the well known transient state space method. As the number of states grows exponentially with the number of Erlang phases, this method is not applicable in general. An approximate method for analyzing regularly structured non-series-parallel graphs is described in [5].

We propose to reduce the number of states by approximating the runtime distribution of each task with one deterministically and one exponentially distributed phase. The parameters of the deterministically distributed phase d and of the exponentially distributed phase λ are obtained from the expected runtime E and the runtime variance V. In order to adapt the first and the second moment we get:

$$\lambda = \frac{1}{\sqrt{V}} \qquad \text{and} \qquad d = E - \sqrt{V} \qquad \left(\text{if } \; E \geq \sqrt{V}\right) \tag{7}$$

If $\sqrt{V} > E$, we approximate the task's runtime with a two phase hyperexponential distribution. The mean runtime of a graph consisting of deterministically and exponentially distributed phases can be obtained with the approximate state space method [18]. The analysis is done in three steps:

1. The runtime distribution of each node is approximated, depending on the mean and the variance of the node, by one deterministically and one exponentially distributed phase or by a two-phase hyperexponential distribution.
2. Serial reduction by adding the expected runtime and the runtime variance of serially connected nodes.
3. The remaining non-series-parallel graph, consisting of deterministically and/or exponentially distributed phases is analyzed by an approximate state space method in the following way [18]:

Every state s of the state space is characterized by the set of tasks $T(s) = \{T_1, T_2, \ldots, T_k, T_{k+1}, \ldots, T_n\}$, which run in parallel in state s.
The phases T_j, $1 \leq j \leq k$ have deterministically distributed service times with parameter d_j.

$d = min(d_1, d_2, \ldots, d_k)$ is the minimum of the deterministic phases and T_m the task with the shortest deterministic runtime in state s. Therefore we have: $d_m = d$.

For $k < j \leq n$ the service time is an exponentially distributed random variable with parameter λ_j.

The system changes from state s to state s'_i, if task T_i is the task which finishes first. The running tasks in s'_i are obtained as $T(s'_i) = \{T'_1, T'_2, \ldots, T'_k, T_{k+1}, \ldots, T_n\} \backslash \{T_i\} \cup S_s(T_i)$ with parameter $d'_j = d_j - E[T_i(s)]$ for $1 \leq j \leq k$ and λ_j for $j > k$. We see, for $j \leq k$ the random variable t'_j is approximated by a deterministically distributed variable with parameter d'_j.
In order to compute the average runtime of the complete program we define the average service time in state s and the probability of T_i being the first which finishes in state s.

$E[T_i(s)]$: The average service time of task T_i in state s under the condition that T_i is the first which finishes.

$p(T_i(s))$: The probability that T_i is the first which finishes in state s.

S : The first state in the state space

Therefore, we can compute the average runtime of the whole parallel program $E[G] = E[S]$ recursively in the following way:

$$E[S] = \sum_{T_i \in T(S)} p(T_i(S)) \big(E[T_i(S)] + E\big[S'_i\big] \big) \tag{8}$$

In order to obtain the branching probability and the expected remaining time in any state we use the Dirac-function $\delta(t)$:

$$\begin{aligned} &\delta(t) := \{^{\infty, \text{ if } t=0}_{0 \text{ , else}} \\ &\int_0^\infty \delta(t)\, dt = 1 \end{aligned} \tag{9}$$

We distinguish between three cases:

Case 1: $1 \leq i \leq k \wedge i \neq m$ (all deterministic phases except the shortest)

$$p(T_i(s)) = 0 \tag{10}$$

Case 2: $i = m$ (the shortest deterministic phase)

$$\begin{aligned} p(T_i(s)) &= P\left(\bigwedge_{j=k+1}^{n} t_i < t_j\right) \\ &= \int_0^\infty f_i(t) \prod_{j=k+1}^{n} (1 - F_j(t))\, dt \\ &= \int_0^\infty \delta(t-d) e^{-\sum_{j=k+1}^{n} \lambda_j t}\, dt \\ &= e^{-\sum_{j=k+1}^{n} \lambda_j d} \\ E[T_i(s)] &= d \end{aligned} \tag{11}$$

Case 3: $k < i \leq n$ (exponentially distributed phases)

$$\begin{aligned} p(T_i(s)) &= P\left(t_i \leq t_m \wedge \bigwedge_{j=k+1 \wedge j \neq i}^{n} t_i \leq t_j\right) \\ &= \int_0^\infty f_i(t)(1 - F_m(t)) \prod_{j=k+1 \wedge j \neq i}^{n} (1 - F_j(t))\, dt \\ &= \frac{\lambda_i}{\sum_{j=k+1}^{n} \lambda_j} \left(1 - e^{-\sum_{j=k+1}^{n} \lambda_j d}\right) \end{aligned} \tag{12}$$

$$\begin{aligned} E[T_i(s)] &= \int_0^\infty t \frac{\partial}{\partial t} P\left(t_i \leq t | t_i < t_m \bigwedge_{j=k+1 \wedge j \neq i}^{n} t_i < t_j\right) dt \\ &= \frac{1 - \left(1 + \sum_{j=k+1}^{n} \lambda_j d\right) e^{-\sum_{j=k+1}^{n} \lambda_j d}}{\left(1 - e^{-\sum_{j=k+1}^{n} \lambda_j d}\right) \sum_{j=k+1}^{n} \lambda_j} \end{aligned} \tag{13}$$

This method is much faster than the exact transient state space analysis. However, analyzing large models, this method may also fail due to the number of states in the state space.

3.3. Bounds for the Mean Runtime

If all these methods fail because of computational costs, we apply methods for obtaining bounds of the expected runtime [6]. There are several published bounding methods. Devroye [3] uses only the first moment and the variance of the tasks' runtime in order to obtain bounds for the mean runtime of the whole program. Lower and upper bounds are also obtained by Yazici-Pekergin and Vincent [21] where the nodes' runtimes are assumed to be independent and identically distributed of type NBUE. The mean service time of the critical path is used as a lower bound. This does not yield good results for high parallelism because only the first moments of the runtime distributions and no parallelism are considered. The upper bound is obtained by replacing the tasks' runtime distribution functions by exponential distribution functions with the same first moment and building the product of the distributions of all paths from a start node to an end node. Again only the first moments are considered and therefore the bounds are poor in general.

Our efforts were oriented towards getting better lower and upper bounds for the mean runtime of parallel programs by using the whole information contained in the tasks' runtime distribution. The three methods implemented in PEPP are based on the same principle. To get an upper bound of the mean runtime we add nodes or arcs to a given non-series-parallel graph in order to make it series-parallel reducible. The mean runtime of the remaining graph

is an upper bound. Removing nodes or arcs from the original graph leads to a lower bound of the mean runtime.

Bounding the Mean Runtime through Adding/Removing Arcs

The first bounding method is based on the fact that modifying a graph by adding/removing arcs leads to a graph with a higher/lower mean completion time [9].

Our goal is to get the series-parallel reducibility by applying a number of such modifications on a given non-series-parallel graph. Obviously, the exclusive adding of arcs leads to an upper bound and by analogy the exclusive removing to a lower bound of the mean runtime.

Generally there are many possibilities to reach the series-parallel reducibility of a graph by adding or removing arcs. Consequently, there is more than one upper and lower bound. The main question now is, how can we obtain the tightest bounds, i.e. the smallest upper bound and the largest lower bound for the mean completion time. To decide which bounds are the best for a given graph we have to compute all the bounds. It is clear that this method is too inefficient, so we decided to solve the problem in a different way. We developed a heuristic algorithm for inserting/removing arcs which enables us to get very tight bounds for the mean runtime in an efficient way.

We will now briefly discuss the algorithm for the upper bound. As mentioned earlier, we add a number of arcs to our graph until it is series-parallel reducible. It is clear that the quality of the bounds depends on the number of the arcs added, i.e. the less the number of new arcs, the better the resulting upper bound.

The algorithm for the upper bound mainly consists of two steps, which are described as follows [12], [6]:

1. Series-parallel reduction of the graph until no more reduction is possible.
2. Search for a pair of nodes which allows us a parallel reduction through adding new arcs. The following conditions must be fulfilled:
   - It is not allowed to add arcs between nodes on the same level.
   - The number of new added arcs which lead to the next series-parallel reduction must be minimal.
   - If this condition is satisfied by more than one pair of nodes we demand that the number of nodes of the resulting graph must be minimal.

These steps are repeated until we have one node left. The resulting mean value of this node's runtime distribution is our upper bound.

The algorithm for the computation of the lower bound is similar to the algorithm described above. Instead of adding arcs we remove arcs. Removing an arc may lead either to a serial reduction or to a parallel reduction. In most cases it is impossible to decide which modification is better. Therefore, we decided to provide two different algorithms for the lower bound. The first one makes only modifications that lead to a serial reduction and the second one only removes arcs which allow a parallel reduction. If there is more than one arc which leads to a serial reduction, we assign the sum of the mean runtime of the start node and the end node as a label to each arc and remove the arc with the smallest label. The maximum of the two different lower bounds is our result for the lower bound. This solution costs two

times more computation time than the algorithm for the upper bound, but we obtain a very good lower bound.

The Method of Shogan

The following method was developed by A.W. Shogan for the evaluation of PERT networks. The main difference between PERT networks and stochastic graph models defined in section 2 is that the tasks are represented by arcs instead of nodes. It is not difficult to transform our model into a PERT network, but here we will explain how we can use this method directly for our graph models.

First, we will introduce an additional notation. Let f_1 and f_2 be two densities and F_1 and F_2 the corresponding distribution functions. Then, we define

$$f_1 \leq f_2 \quad :\leftrightarrow \quad (\forall t \geq 0)(F_1(t) \geq F_2(t))$$

The same inequality between the corresponding first moments follows from the above definition of the relation '$\leq$' [9].

Now, we will describe the method of Shogan. The proof of the correctness of this method is given in [17].

Let $G = (V, E, \underline{T})$ with $V = \{v_1, \ldots, v_n\}$ and $E \subset V \times V$ be a stochastic graph, F_{T_i} the distribution function of each node, and f_{T_i} the corresponding density. Then, we can recursively compute the lower bound $g^<$ and the upper bound $g^>$ for the whole distribution density g as follows [9, 17]:

$$\begin{aligned} g^< &= \text{shogan}\Big\{g_j^< \mid s(v_j) = \emptyset\Big\} \\ g_i^< &= \begin{cases} f_i & \text{if } p(v_i) = \emptyset \\ \text{conv}\Big(\text{shogan}\Big\{g_j^< \mid v_j \in p(v_i)\Big\}, f_i\Big) & \text{else} \end{cases} \end{aligned} \tag{14}$$

$$\begin{aligned} g^> &= \max\Big\{g_j^> \mid s(v_j) = \emptyset\Big\} \\ g_i^> &= \begin{cases} f_i & \text{if } p(v_i) = \emptyset \\ \text{conv}\Big(\max\Big\{g_j^> \mid v_j \in p(v_i)\Big\}, f_i\Big) & \text{else} \end{cases} \end{aligned} \tag{15}$$

where

$$\begin{aligned} \min(F,G)(t) &= \begin{cases} F(t) & \text{if } F(t) < G(t) \\ G(t) & \text{else} \end{cases} \\ \text{shogan}(f,g)(t) &= \frac{d}{dt}\min(F,G)(t) \\ \text{conv}(f,g)(t) &= \int_0^t f(t-x)g(x)dx \\ \max(f,g)(t) &= f(t)G(t) + F(t)g(t) \\ \max(f_1, f_2, \ldots f_n) &= \max(f_1, \max(f_2, \ldots, \max(f_{n-1}, f_n))) \\ \text{shogan}(f_1, f_2, \ldots f_n) &= \text{shogan}(f_1, \text{shogan}(f_2, \ldots, \text{shogan}(f_{n-1}, f_n))) \end{aligned} \tag{16}$$

The first moments of the densities $g^>, g^<$ now build upper and lower bounds of the mean runtime.

Obviously, these bounds are easier to compute, i.e. with lower computational effort than with the first method. Unfortunately, Shogan's lower bound is not as good as the bound obtained by removing arcs. The upper bound of Shogan, however, deserves special consideration. It is noteworthy that in some cases Shogan's upper bound is better than the least upper bound obtained by adding arcs. According to our experience the upper bound obtained by adding arcs is better than Shogan's upper bound in most cases.

The Upper Bound of Dodin

This method was also developed for the evaluation of PERT networks [4]. The main idea behind this method is to modify the PERT network through duplicating some arcs until it is series-parallel reducible. Applying this method to stochastic graph models means that nodes are duplicated. The method is based on the fact that duplicating nodes always leads to an upper bound of the mean runtime.

At the first view this looks like the upper bound of Shogan, where the max operator can be considered as the duplication of nodes, too. However, there are two big differences between these methods. The first difference lies in the duplication process. Computing Shogan's upper bound, we observed that during the duplication of a node we duplicated all predecessor nodes, too. The duplication process of Dodin, however, doesn't have this disadvantage.

Another difference is the non-determinism of Dodin's method. There are many ways to bring a non-series-parallel graph into a reducible form.

Now we will discuss the duplication process in more detail. Therefore we need the definition of nodes which can be duplicated.

Definition: The node v can be duplicated by adding v', if either

1. $|p(v)| \leq 1 \wedge |s(v)| \geq 2$ (case 1), or
2. $|s(v)| \leq 1 \wedge |p(v)| \geq 2$ (case 2)

The duplication of a node v can be processed as follows:

case 1: For the duplication of node v choose one of the successor nodes v_i of v, and add an arcs from the predecessor arc of node v to node v' and arc (v', v_i). Then, remove arc (v, v_i).

case 2: Choose one predecessor v_i of v, and by analogy add the arc (v_i, v') add an arc from v' to the successor node of node v. Then, remove arc (v_i, v).

Finally a few words about our implementation of this method. We compute the upper bound of Dodin as follows:

- Series-parallel reduction
- Identify and duplicate all nodes which can be duplicated and integrate the new nodes by adding two new arcs into the graph and removing one arc.
- Repeat these two steps until one node remains.

A more detailed description of the different methods and a comparison is given in [11], [6].

The Analysis of Hierarchical Graphs

Both, graph construction and graph analysis can be done hierarchically. Choosing a hierarchical evaluation method means that the subgraphs of level $i-1$ are analyzed and the

distribution functions or the mean runtimes are inserted into the graph of level i. Applying the approximate state space method each subgraph is replaced by one deterministically distributed phase and we obtain a lower bound for the mean runtime. With the series-parallel-reduction method each subgraph is replaced by a lower or an upper bound inserting the best bounds which are obtained by the different methods. Therefore, we use the advantages of the different methods to get the best bounds for the mean runtime.

If a given graph already has a series-parallel structure, the bounding methods always lead to exact results.

4. Numerical Results

In this section we show the quality of the de-approximation and of the bounding methods dependent on the distribution functions of the task runtimes.

4.1. The Accuracy of the de-approximation

Many simulations were done with the petri net tool GSPN [1] in order to compare the approximated runtimes with simulation results. The differences between approximated and simulation results lie between 0 and 5 percent of the greater result in all investigated examples. It is readily seen that the introduced method in section 3.2 is exact, if a model consists of only deterministically or only exponentially distributed task runtimes.

The graph in figure 2 shows a model of a parallel program for multigrid solutions of a partial differential equation on a shared memory multiprocessor [7]. These graphs are regularly structured and can be described by the height n and the width m. The data dependencies constrain that a task on level i must wait until its own predecessor and the predecessor of its left and right neighbor have finished the computation on level $i-1$ (neighbor synchronization).

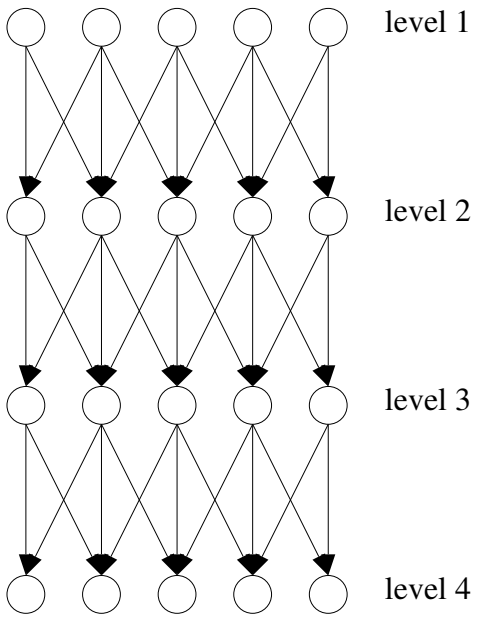

Figure 2: Model for a 4×5 neighbor synchronization

We compare approximated results with exact values and simulation results for the model in figure 2. As a workload we assume identically distributed runtimes for all tasks. The

accuracy of the proposed approximation is tested by workload distributions of type Erlang. In order to be able to compare results, λ and k have been chosen in order to get the first moment constant and the variance different.

task runtime (workload)	exact		de-approximation	simulation result
	number of states	value		
$E_1(0.2)$	260	40.80	40.80	40.64 ± 0.25
$E_2(0.4)$	6052	34.72	34.70	34.35 ± 0.44
$E_5(1)$	too high computation costs		29.24	29.26 ± 0.39
$E_{10}(2)$			26.47	26.67 ± 0.51

Table 1: Comparison of exact, approximated, and simulation results

The column with *de-approximation* contains the computed runtime for approximated models. The $E_k(\lambda)$ distributed task runtimes are approximated by one deterministically and one exponentially distributed phase. We get exact results only in the case, the task runtimes are modeled by $E_1(\lambda)$ or $E_2(\lambda)$ distributed random variables. In the case of $E_5(\lambda)$ and $E_{10}(\lambda)$ distributed task runtimes the computation costs of a markovian analysis are too high (number of states in the state space). We can compare the approximation results with simulation results to get information about the quality of the approximation method. The task runtimes in the simulation models are also approximated by one deterministically and one exponentially distributed phase. Table 1 shows that the approximated runtimes lie in the 0.99 confidence interval of the simulation results.

4.2. Comparing the Bounding Methods

Evaluating neighbor structures we observed that the quality of the computed bounds depends on the size of the given structure and the task runtime distributions. In order to demonstrate these dependencies we will first consider neighbor-$(8 \times m)$–structures with $m \in \{4, 6, 8, 10, 12\}$. All task runtimes are identically $\mathrm{E}_{10}(2)$ distributed, where $\mathrm{E}_{10}(2)$ denotes the Erlang-10 distribution with rate $\lambda = 2$ s^{-1}. The results are shown in figure 3. The extensions .u and .l stand for upper bounds and lower bounds, respectively Klein is the abbreviation of Kleinöder.

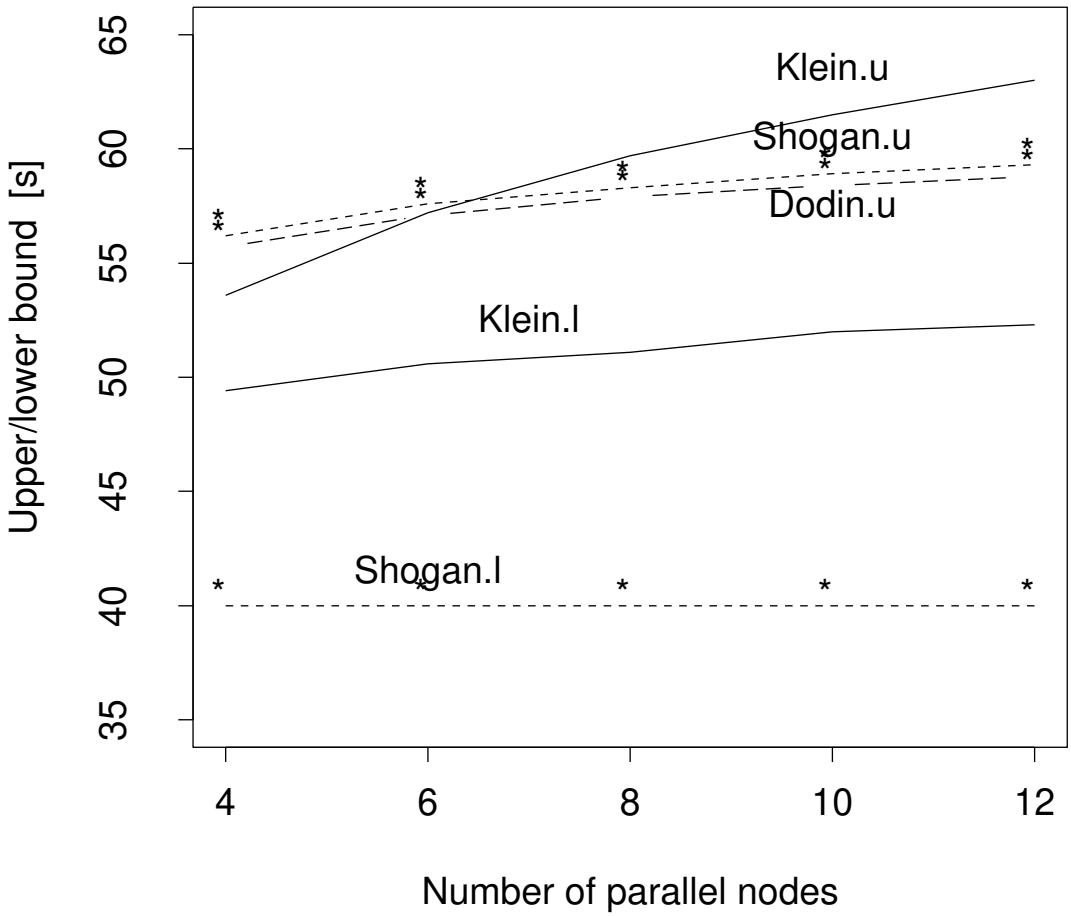

Figure 3: Bounds for the mean runtime

We observe that Shogan's lower bound is always less than the lower bound obtained by removing arcs. Unfortunately, for $m > 6$ the upper bound obtained by adding arcs is not as good as the other two upper bounds. Adding arcs to a neighbor-structured graph leads only to a series-parallel graph if each node is connected with every node of the previous and the next level. Accordingly, for the series-parallel reducibility of a neighbor-$(n \times m)$-structure it is necessary to add $(n-1)(m^2-3m+2)$ new arcs. With Dodin's method, however, it is possible to reach the series-parallel reducibility through $(n-1)(2m-2)$ new nodes. Hence follows that for $m > 4$ we have to add more arcs than nodes needed for the series-parallel reducibility. Nevertheless, we observe that for $n = 8$ and $m = 4$, Klein.u is less than Dodin.u, although for both methods 42 new arcs respectively nodes are necessary. The reason for this is that the duplication of one node leads, according to our experience, to a higher mean runtime of the whole graph than adding one single arc.

Table 2 contains the computation costs T_c for obtaining results with the de-approximation and the different bounding methods. Exact results are not available for models of this dimension. We see that the de-approximation can be used only for $m < 6$ due to the state

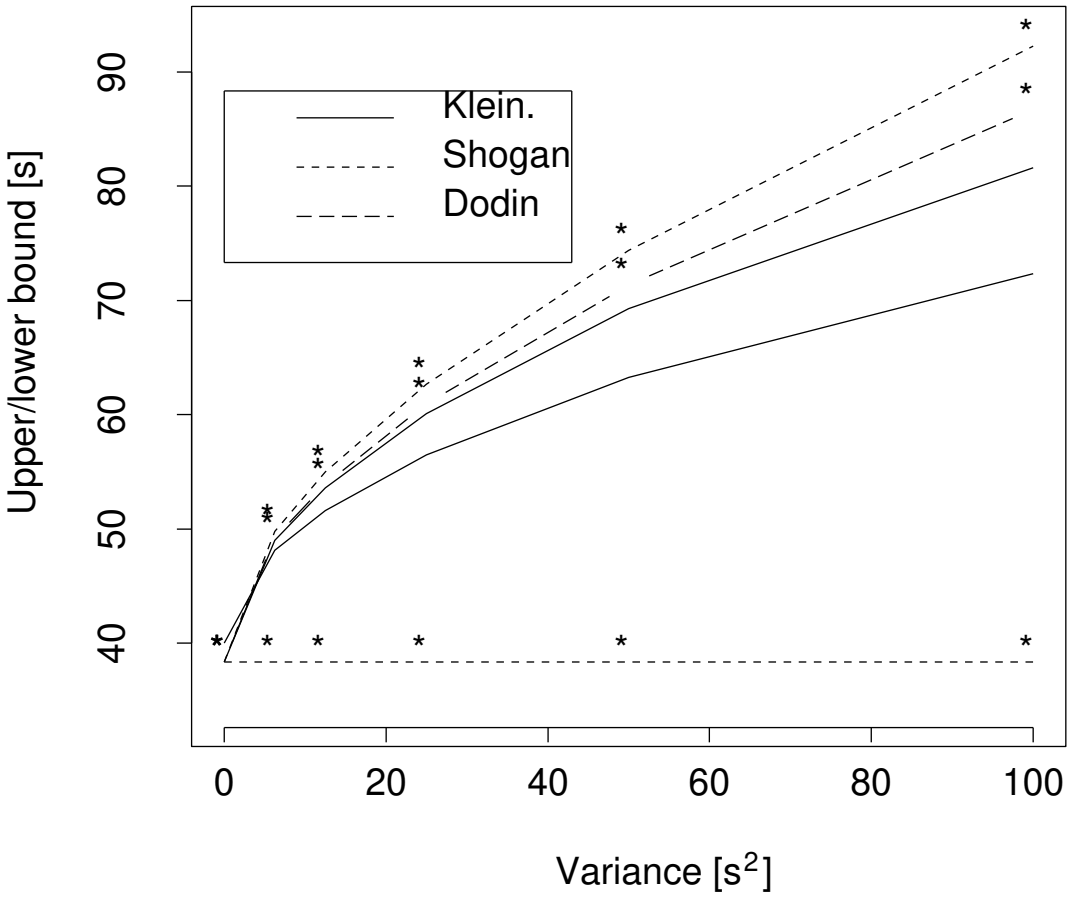

Figure 4: Bounds for the mean runtime

space explosion with increasing m. The best lower and upper bound is typed in bold style. Using the bounding methods, the computation costs are small and grow only polynomially with m.

m	de-approximation			bounding methods									
				Klein				Shogan				Dodin	
				lower		upper		lower		upper		upper	
	result	# states	T_c [s]	result	T_c [s]	result	T_c [s]	result	T_c [s]	result	T_c [s]	result	T_c [s]
4	51.1	4252	15	**49.4**	2	**53.6**	1	40.0	1	56.2	1	54.8	1
5	52.4	62329	1430	**49.2**	3	**55.6**	1	40.0	1	57.0	2	55.8	1
6				**50.6**	3	57.2	2	40.0	1	57.6	2	**56.3**	2
8				**51.1**	5	59.7	6	40.0	2	58.3	3	**57.2**	2
10				**52.0**	9	61.5	10	40.0	2	58.9	3	**57.8**	3
12				**52.3**	14	63.0	18	40.0	2	59.3	4	**58.2**	3

Table 2: Comparison of the computation costs

Now, we will consider the task runtimes. As mentioned earlier, the task runtime distributions play an important role for the quality of the bounds. We observed that the computed bounds are tighter when we use distributions with low variance. This dependency is shown in figure 4. We consider neighbor-(4×4)-structures where the expected runtime of each task is 10 ms.

While the quality of all bounds is low for very high variances, we observe that Klein.u is distinctly better than the other two upper bounds. If there is no variance, then all node runtimes have the same deterministic distribution and the bounds obtained by all methods are equal to the exact result.

5. Conclusions and Prospects

This paper gives a survey of modeling possibilities and analysis techniques implemented in the analysis tool PEPP. The approximate state space analysis allows us to obtain very accurate results for the mean runtime of graphs with a non-series-parallel structure. The series-parallel reduction based on numerical distribution functions leads to the runtime distribution of the modeled program in a very efficient way. Applying this method to non-series-parallel graphs, we get lower and upper bounds for the mean runtime and the runtime variance.

For further research it would be of interest to combine the advantages of the different bounding strategies in one graph level, insertion of arcs and insertion of nodes, in order to get better bounds in less computation time.

References

[1] G. Chiola. *GreatSPN Users' Manual*, 1987.

[2] P. Dauphin, R. Hofmann, R. Klar, B. Mohr, A. Quick, M. Siegle, and F. Sötz. ZM4/SIMPLE: a General Approach to Performance–Measurement and –Evaluation of Distributed Systems. In T.L. Casavant and M. Singhal, editors, *Readings in Distributed Computing Systems*. IEEE Computer Society Press, 1992.

[3] L.P. Devroye. Inequalities for the Completion Times of Stochastic PERT Networks. *Math. Oper. Res.*, 4:441–447, 1980.

[4] B. Dodin. Bounding the Project Completion Time Distributions in PERT Networks. *Operations Research*, 33(4):862–881, 1985.

[5] G. Fleischmann. *Performance Evaluation of Parallel Programs for MIMD-Architectures: Modeling and Analyses (in German)*. Dissertation, Universität Erlangen–Nürnberg, 1990.

[6] F. Hartleb and V. Mertsiotakis. Bounds for the Mean Runtime of Parallel Programs. In R. Pooley and J. Hillston, editors, *Sixth International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 197–210, Edinburgh, 1992.

[7] R. Hofmann, R. Klar, N. Luttenberger, B. Mohr, and G. Werner. An Approach to Monitoring and Modeling of Multiprocessor and Multicomputer Systems. In T. Hasegawa et al., editors, *Int. Seminar on Performance of Distributed and Parallel Systems*, pages 91–110, Kyoto, 7–9 Dec. 1988.

[8] R. Klar, A. Quick, and F. Sötz. Tools for a Model–driven Instrumentation for Monitoring. In G. Balbo, editor, *Proceedings of the 5th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 165–180. Elsevier Science Publisher B.V., 1992.

[9] W. Kleinöder. *Stochastic Analysis of Parallel Programs for Hierarchical Multiprocessor Systems (in German)*. Dissertation, Universität Erlangen–Nürnberg, 1982.

[10] J.J. Martin. Distribution of the Time through a Directed, Acyclic Network. *Operations Research*, 13(1):46–66, 1965.

[11] V. Mertsiotakis. Erweiterung des Graphanalyseprogramms SPASS und Integration in die X–Window–Umgebung von PEPP. Internal study, Universität Erlangen–Nürnberg, IMMD VII, 1991.

[12] V. Mertsiotakis. Extension of the Graph Analysis Tool SPASS and Integration into the X-Window Environment of PEPP (in German). Internal study, Universität Erlangen–Nürnberg, IMMD VII, 1991.

[13] H. Pingel. Stochastic Analysis of seriesparallel programs (in German). Internal study, Universität Erlangen–Nürnberg, 1988.

[14] R. Sahner. *A Hybrid, Combinatorial Method of Solving Performance and Reliability Models*. PhD thesis, Dep. Comput. Sci., Duke Univ., 1986.

[15] R. Sahner and K. Trivedi. Performance Analysis and Reliability Analysis Using Directed Acyclic Graphs. *IEEE Transactions on Software Engineering*, SE-13(10), October 1987.

[16] R. Sahner and K.S. Trivedi. SPADE: A Tool for Performance and Reliability Evaluation. In N. Abu El Ata, editor, *Modelling Techniques and Tools for Performance Analysis '85*, pages 147–163. Elsevier Science Publishers B.V. (North Holland), 1986.

[17] A.W. Shogan. Bounding Distributions for a Stochastic PERT Network. *Networks*, 7:359–381, 1977.

[18] F. Sötz. A Method for Performance Prediction of Parallel Programs. In H. Burkhart, editor, *CONPAR 90–VAPP IV, Joint International Conference on Vector and Parallel Processing. Proceedings*, pages 98–107, Zürich, Switzerland, September 1990. Springer–Verlag, Berlin, LNCS 457.

[19] F. Sötz and G. Werner. Load Modeling with Stochastic Graphs for Improving Parallel Programs on Multiprocessors (in German). In *11. ITG/GI–Fachtagung Architektur von Rechensystemen*, München, 1990.

[20] A. Thomasian and P.F. Bay. Analytic Queueing Network Models for Parallel Processing of Task Systems. *IEEE Transactions on Computers*, C-35(12):1045–1054, December 1986.

[21] N. Yazici-Pekergin and J.-M. Vincent. Stochastic Bounds on Execution Times of Parallel Programs. *IEEE Transactions on Software Engineering*, 17(10):1005–1012, October 1991.