Stochastic Modeling of Scaled Parallel Programs[*]

Allen D. Malony
Department of Computer and Information Science
University of Oregon
Eugene, OR 97403, USA

Vassilis Mertsiotakis
University of Erlangen-Nürnberg
IMMD VII, Martensstr. 3
91058 Erlangen,
Germany

Andreas Quick
Thermo Instrument Systems
Frauenauracher Str. 96
91056 Erlangen,
Germany

Abstract

Testing the performance scalability of parallel programs can be a time consuming task, involving many performance runs for different computer configurations, processor numbers, and problem sizes. Ideally, scalability issues would be addressed during parallel program design, but tools are not presently available that allow program developers to study the impact of algorithmic choices under different problem and system scenarios. Hence, scalability analysis is often reserved to existing (and available) parallel machines as well as implemented algorithms.

In this paper, we propose techniques for analyzing scaled parallel programs using stochastic modeling approaches. Although allowing more generality and flexibility in analysis, stochastic modeling of large parallel programs is difficult due to solution tractability problems. We observe, however, that the complexity of parallel program models depends significantly on the type of parallel computation, and we present several computation classes where tractable, approximate graph models can be generated.

Our approach is based on a parallelization description of programs to be scaled. From this description, "scaled" stochastic graph models are automatically generated. Different approximate models are used to compute lower and upper bounds of the mean runtime. We present evaluation results of several of these scaled (approximate) models and compare their accuracy and modeling expense (i.e., time to solution) with other solution methods implemented in our modeling tool PEPP. Our results indicate that accurate and efficient scalability analysis is possible using stochastic modeling together with model approximation techniques.

1 Introduction

In order to implement portable and efficient parallel programs which will also have good performance scalability, parallelization choices must be tested for many systems and problem testcases. Since a program's behavior could vary for different problem sizes and different numbers of processors, a systematic test method is desired. Presently, scalability testing is often deferred until after a program has been implemented, restricting its application to existing system environments. Ideally, scalability issues would be addressed during parallel program design, allowing the developer to explore the performance scaling characteristics of different algorithm alternatives before implementation commitment. To do so, however, requires a scalability analysis approach that allows quantitative statements to be made relating the parallelization properties of programs and the critical execution factors of target machines to scaled performance behavior. Even if the program properties and machine factors can be accurately represented, the available solution techniques are limited in their ability to analyze the performance effects from program/machine interactions.

Modeling parallel programs with stochastic models like graph models [14] or Petri net models [2] is a well–known, proven method to analyze a program's dynamic behavior. It can be used to predict the program's runtime [17], and, by changing model parameters, help to understand the program's general performance behavior, to investigate reasons for performance bottlenecks, or to identify program errors. In considering the use of stochastic modeling for the analysis of performance scalability, we show that speedup values can be computed from the predicted runtimes of model instances for different numbers of processors and problem sizes. Hence, a model–based analysis need not be restricted to existing systems, and a stochastic modeling tool can be expected to provide an environment for systematic study of tradeoffs in parallelization strategies and execution time functions (representing the computation, synchronization, and communication properties of different target execution environments) that govern scaled performance characteristics.

However, a systematic method for analyzing scaled programs based on modeling presents several challenging problems. First, in general, it must be possible to create models for different configurations and topologies of a parallel system as well as for different problem sizes. One approach might be to develop a model generator which automatically creates multiple "scaled" models by extending a basic "generic" model of the program to be analyzed[1]. Adopting this strategy, a second problem must be addressed – how is the generic model represented. Our solution is the *P*arallelization *D*escription *L*anguage (PDL) [12], developed for describing the structure of parallel programs, the parallelization scheme for each parallel program part, and various aspects of a program's runtime behavior. A third problem lies in producing scaled models from generic

[1]Note, this "generator" approach could be undertaken with different modeling techniques, including Petri net models [19] and Queuing Models [10].

representations, since model complexity, tractability, and solution accuracy must be considered. The key is to find model generation methods which produce approximately accurate models of scaled performance behavior, but that do not exceed the solution capabilities of stochastic modeling tools. With respect to runtime distributions, we agree with Adve and Vernon [1] that the use of exponentially distributed task execution times to permit tractability is not representative of actual parallel program behavior. Instead, we promote the use of efficient techniques for analyzing parametrical distributions (especially Erlang distributions which can have arbitrary low variance) or numerical distributions (obtained from monitoring existing systems) for tractable solution to scaled models, in contrast to solution simplifications afforded by a deterministic model [1]. Lastly, systematic methods for scalability study must address the question of automatic testcase analysis. The most appropriate approach would be to integrate performance scalability into stochastic modeling tools.

Our goal in this paper is to demonstrate the application of stochastic modeling to the scalability analysis of parallel programs. We do so by presenting techniques for parallel program representation, generic model transformation, and scaled model analysis that allow bounding performance results to be derived. To evaluate the efficacy of our techniques, we have integrated model generation and scalability analysis into our tool PEPP (*P*erformance *E*valuation of *P*arallel *P*rograms) [7]. In addition to gaining access to the efficient solution techniques supported by PEPP, this integration has also allowed us to evaluate tool requirements for automatic scalability study.

The remainder of the paper is organized as follows. In §2, the concept of model–based analysis of scaled parallel programs is introduced. Here, we also discuss the evaluation of stochastic graph models using different bounding techniques. The automatic creation of scalability models is addressed in §3. Different parallel computation classes are described, and it is shown how scaled approximate models of those classes are derived. In §4, we discuss the use of PEPP for scalability analysis. A detailed example of scalability analysis for a neighbor synchronization computation is presented where different solution techniques are compared. Our results indicate that accurate solutions for scaled approximate models can be obtained. Finally, in §5, we remark on several open issues concerning the general validity of stochastic modeling methods for studying performance scalability.

2 The methodology

2.1 Concept for automatic scalability analysis

To avoid the complexity of developing parallel program models for target parallel systems from scratch, Herzog proposed a "three step methodology" [8]. Instead of creating a single, monolithic model for each combination of workload, machine configuration, and load distribution, a *workload model* is developed independent of its implementation concerns; the *machine model* is also developed separately. The *system model* is then obtained by mapping the workload model onto the machine model. The combined model reflects the dynamic, mapped program behavior and shows how system resources are used (Figure 1).

Our approach for scalability analysis extends this base methodology to generate multiple system models that re-

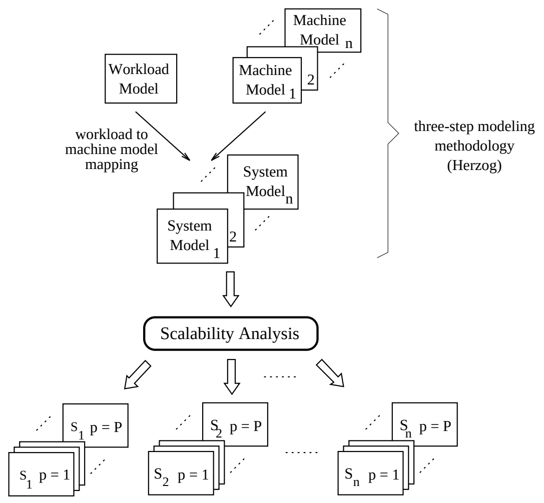


Figure 1: Concept for automatic scalability analysis

flect varying degrees of parallelism and problem sizes. In contrast to Herzog's approach, the choice of number of processors is separated from the machine model; the mapping of the workload model onto the machine model must be realized for each number of processors separately, creating several *scaled models*. In our approach, the machine model only describes synchronization mechanisms and performance distributions for each machine component.

The challenge of scalability modeling following this methodology is to define systematic techniques for generating scaled models, from generic workload and machine models, that can accurately capture scaled performance behavior. Two issues must be addressed. First, parallelization schemes must be well understood in order for the automatic mapping of tasks to processors to occur. This requires some representational form to be defined that identifies parallelization characteristics for different classes of computations. Our solution, PDL, is discussed in [12]. The second issue – scaled model generation – is, perhaps, more problematic. Some parallelization schemes can easily lead to scaled models whose exact solutions are prohibitive in computational requirements. Although modeling techniques have been developed that are "largeness tolerant" [18] (i.e., can deal to some extent with model complexity), the process of creating a correct and solvable exact model is non–trivial. To overcome these exact model generation and evaluation problems, we must develop approximation techniques that simplify model creation and reduce the complexity of model analysis. Clearly, the scaled models should not be so simple that analysis accuracy is sacrificed. Our approach is to create approximate models that bound scaled performance, and to use more accurate modeling, where efficient solutions are possible, to tighten, where needed, the performance range.

In general, approximate model generation is not easy due to problems such as task dependencies, scheduling, and synchronization. In addition to the generation of scaled models, task density functions must be derived that capture the performance effects of interactions between scaled parallel tasks. It is our aim to implement not only scalable, but also portable parallel programs. Further evaluation complexity is introduced since we must consider the desire for different machine models to be applied in scalability study. By mapping the workload model onto n different machine models we obtain at least n different system models, each generating a scaled model set. Our approximation and solution techniques must be accurate as well as efficient to make model–based scalability analysis an acceptable practice.

Scalability analysis can be carried out with any method using discrete event models. In the remainder of this paper we examine scalability analysis for stochastic graph models. Our approach using the description language PDL could easily be undertaken with Petri net models as in [19].

2.2 Evaluation of stochastic graph models

Stochastic graph models have been used extensively to model and to analyze the behavior of parallel programs [15]. The execution order of program activities, their runtime distribution, and branching probabilities can be represented using stochastic graph models. Besides modeling algorithmic properties, graph models can also be used to model the mapping onto a parallel machine, which is a prerequisite for scalability analysis. A parallel program is modeled by a graph, $G = (V, E, T)$, which consists of a set of nodes, V, representing program tasks and a set of directed edges (arcs), $E \subset V \times V$, modeling the dependencies between the tasks. To each program task v_i a random variable $T_i \in T$ is assigned which describes the runtime behavior of v_i ($T_i, i = 1, \ldots, n$, are assumed to be independent random variables). To evaluate stochastic graph models, various methods can be applied.

State space analysis
If the tasks' runtime distributions are given by general Erlang functions, we can use well–known transient state space analysis. However, as the number of states grows exponentially with the number of Erlang phases, the number of nodes, and the number of task dependencies, this method is not applicable in general, especially for scaled models of parallel programs which often consist of hundreds of nodes. If these large models are not structured in a series–parallel manner, exact evaluation methods fail because of excessive computation times and memory requirements (state space explosion). Therefore, we propose to reduce the number of states by approximating the runtime distribution with one deterministically and one exponentially distributed phase (*de-approximation*). The parameters of the deterministically distributed phase d and of the exponentially distributed phase λ are obtained from the expected runtime and the runtime variance (see [17] for details).

Using the approximate state space analysis, another disadvantage of the classical state space analysis can be eliminated: approximate state space analysis can deal with numerical distributions obtained from monitoring. But this method still fails for scalability analysis when modeling a high degree of parallelism where non–trivial interprocessor dependencies are present.

Series–parallel reduction
Graph models which have a series–parallel structure can easily be evaluated using the operators *series reduction* (i.e., convolution of two density functions) and *parallel reduction* (i.e., product of two distribution functions) to reduce the graph to one single node. The runtime distribution of the remaining node gives the runtime distribution of the whole graph model. Series–parallel reduction tolerates largeness because model solution does not require creating a state space [18].

However, some models of parallel programs do not have a series–parallel structure. In these cases, techniques must be developed to transform arbitrary models into models which are series–parallel reducible. If, after applying this transformation, the computed runtime of the transformed models are still considered to be accurate, viable methods for bounding the mean runtime can be formulated.

Bounding methods
There are several bounding methods [5, 11, 16, 21] that can be used to create series–parallel reducible graphs for which bounds on the mean runtime of the program's execution time can be computed. The methods add or delete nodes or arcs to the original graph. If nodes or arcs are added, the mean runtime of the new graph is an upper bound. Removing nodes or arcs from the original graph leads to a lower bound on the mean execution time. In the following, we explain two different methods:

- **The method of Kleinöder**
  Modifying a graph by adding/deleting arcs leads to a graph representing a higher/lower mean execution time [11]. The insertion of arcs increases the interprocessor synchronization causing higher execution times. The deletion of arcs removes synchronization dependencies implying lower execution times. Generally there are many possibilities to reach the series–parallel reducible form of a graph by adding or deleting arcs. Consequently, there is more than one upper or 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). In our modeling tool PEPP (see §4) a heuristic approach is implemented.

- **The method of Dodin**
  This method can be used to obtain an upper bound only [5]. The idea behind this method is to modify the graph model by duplicating nodes. Like the method of Kleinöder, there are many ways to bring a non–series–parallel graph into a reducible form.

Note, these bounding methods operate on an exact model of the parallel program. The problem of creating an exact model remains. We develop "scaled approximate models" using the method of Kleinöder to derive tractable series–parallel reducible models without requiring a large scaled exact model to be created. In §4, we show how the method of Dodin can be used to come to a better upper bound for neighbor synchronization computation classes, but at the cost of a slightly more complex derivation.

3 Scalability models for parallel programs

Our goal with automatic scalability analysis is to make it possible for modeling tools to be applied to scaled versions of parallel programs where it is the number of processors or size of problem or both that are changing. In [12], we approached this problem by considering approximate scaled models for different computation classes. In addition to the trivial case of scaling n independent, identically distributed tasks, the parallel computation classes with dependent tasks shown in Figure 2 were considered. The resulting scaled models for dependent task computation have a more complex structure than the independent task models due to the synchronization arcs in the task graph. In general, task dependencies can be arbitrary. In practice, computations with regular (and often static) task dependencies are quite common in real–world applications.

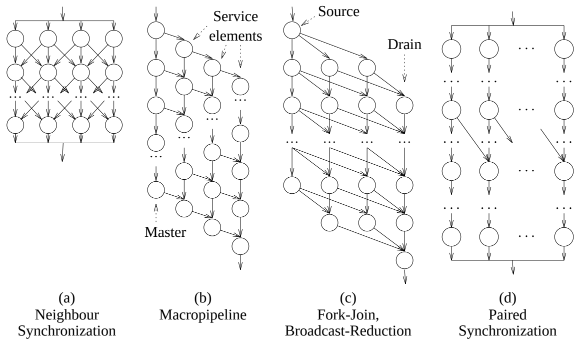

Service elements
Source
Drain
Master
(a) Neighbour Synchronization
(b) Macropipeline
(c) Fork-Join, Broadcast-Reduction
(d) Paired Synchronization

Figure 2: Parallel computation classes (dependent tasks)

Before looking at a specific case, it is instructive to consider what problems we might encounter. Computation classes are best defined by the pattern of task interaction; that is, dependency constraints. The problem size often translates into the number of tasks represented in the computation graph and the number of iterations of the basic graph structure (i.e., phases of the computation). The task density functions are rarely random: either they are related by the type of algorithm, or the same set of functions is used several times because the computation repeats. When generating a scaled model, we must try to determine some property of the computation class that allows us to transform the generic model, representing the detailed (exact) computation, to a tractable graph model.

Structurally, the scaled model should be of a form that is series–parallel reducible in order to allow model evaluation techniques that avoid state space analysis. Thus, not only the size of the generic model, but also the dependency structure must be transformed. For instance, we would prefer that the task graph of the scaled model be a function of the number of processors, rather than the number of "scaled" generic tasks. The trick will be to perform model scaling in a way that does not sacrifice modeling accuracy. Scaled models are created using the knowledge obtained from evaluating graph models with bounding methods. That is, scaled models can be created which allow efficient techniques to be applied. However, because performance scalability is intimately tied to parallel task interactions, reducing the detail at which these interactions are modeled in order to allow tractable solutions risks the loss of performance predictability.

3.1 Neighbor synchronization

To illustrate our general graph scaling approach, consider a neighbor computation structure that can be used to model many iterative solution methods for linear equation systems (Figure 2(a)). The main characteristic of this computation class is that a processor starts the i-th iteration only after the $(i-1)$-th iteration has finished on its neighbor processors [9]. (Although Figure 2(a) shows only two neighbor processes, in general, the number of neighbor tasks can be greater than two.)

The generic graph model for the parallel computation class with neighbor synchronization is shown in Figure 3. If we were to represent each task and dependency in the generic model in the scaled model, the graph size and complexity would be unmanageable. However, a simple graph transformation that results in an upper bound model collapses the neighbor synchronization between iterations to a single barrier, as shown in Figure 3. Once this is done, it is easy to identify that the tasks at each iteration are independent and can be modeled by the techniques for scaling independent tasks [12].

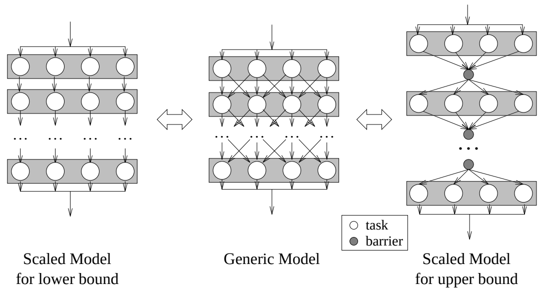


Figure 3: Scaling neighbor synchronization models

This conservative synchronization approximation is a quick way to reduce scaled graph complexity. However, depending on the properties of per iteration task execution time distribution, scaled graph models with tighter upper bound approximations can be generated. For instance, if task density functions are identical per iteration, we might choose to model several successive iterations exactly, separating iteration "clusters" with barrier synchronizations when the analysis complexity becomes too great. Alternative techniques can also be used when considering the inclusion of communication times for data transfer between neighbors per iteration. The main point is that the regular computational structure of the neighbor synchronization classes affords us a certain degree of flexibility in generating approximate graph structures.

3.2 Fork-join, broadcast-reduction

Fork-join models are characteristic of computations where a source periodically generates jobs that spawn tasks to be completed with a drain node collecting results (Figure 2(c)). The graphs that result also have many similarities to graphs generated from computations that involve a sequence of broadcast and reduction operations. Such graphs are typical of linear algebra computations.

As an example, consider the generic LU-decomposition graph in Figure 4; the graph shown here is for a 6 x 6 matrix. Given a large matrix, the graph would consist of several thousands of nodes, making certain solution techniques computationally intractable [19]. However, we can transform the generic model to simpler scaled models. Again, our standard technique can be applied in this case by identifying independent tasks at different iteration levels. However, because of the implicit fork-join nature of the computation, its explicit representation in the scaled models is less likely to lead to modeling inaccuracies.

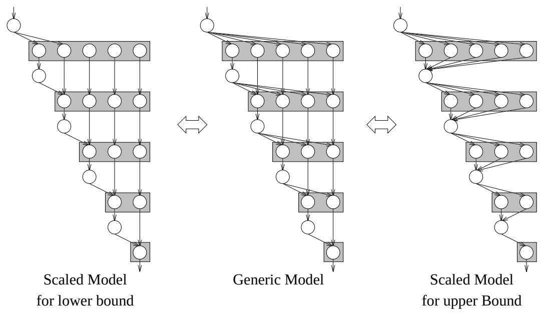
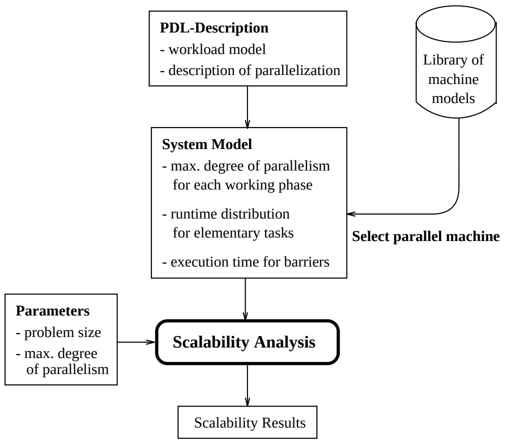


Figure 4: Scaling fork-join, broadcast-reduction models

4 Automatic scalability analysis with the tool PEPP

PEPP (*P*erformance *E*valuation of *P*arallel *P*rograms) [7] is a modeling tool for analyzing stochastic graph models. It provides various methods for model evaluation in order to compute the mean runtime and the runtime distribution of a program represented in the model. PEPP supports efficient solution methods including a series–parallel structure solver, an approximate state space analysis [17], and bounding methods to obtain upper and lower bounds of the mean runtime [7]. In order to model measured runtimes, numerical runtime distributions, which may be obtained from monitoring, are allowed in all three cases. PEPP also incorporates the M^2–cycle methodology [13] for generating a performance model from a functional model. Here, a functional model of a program to be transformed into a performance model is used for event selection, automatic program instrumentation, and event trace evaluation.

PEPP can be used for automatic scalability analysis by creating multiple stochastic graph models based on the PDL program description (Figure 5). After the maximal degree of parallelism P and the problem size are specified, PEPP creates P different graph models. These models are then evaluated and speedup values are calculated from the predicted execution times. Results are presented in a speedup chart.



PDL-Description
- workload model
- description of parallelization
Library of machine models
System Model
- max. degree of parallelism for each working phase
- runtime distribution for elementary tasks
- execution time for barriers
Select parallel machine
Parameters
- problem size
- max. degree of parallelism
Scalability Analysis
Scalability Results

Figure 5: Scalability analysis using PDL

As done with performance monitoring, modeling can classify different parts of the program in order to obtain a detailed scalability profile (loss analysis [3, 4]). The relative influence of the different program phases on the program's execution time can be determined. For the latter, the execution time of all program phases not considered should be set to deterministic runtimes with mean value 0. Using this technique, speedup values can be computed for only the selected program parts.

4.1 Example: scalability analysis of a iterative algorithm

In this section, we give an example for a model-based scalability analysis of a iterative algorithm which might be a part of a larger numerical computation. The analyzed algorithm belongs to the neighbor computation class (see Figure 3). In order to model real program behavior, the execution of each task in our example is assumed as a Erlang–40 distribution. The domain to be calculated is a matrix with 40 rows and n columns, where $n \in \{10, 50, 100, 200\}$. This example shows the usefulness of model–based scalability analysis as well as the problems encountered when modeling parallel systems. Using PEPP, scalability analysis can be carried out in three different ways.

1. **Accurate modeling with state space analysis**
   Even when applying approximate state space analysis using the de-approximation method, only small models can be evaluated. Increasing parallelism leads to a state space explosion and to unsolvable models. The largest model which can be evaluated using this method is one for a 10×40 matrix modeling only two iterations for 4 processors. For this model, the state space is about 160,000 states, and model solution takes about 9 hours on a HP 715 workstation. Since evaluating large models (especially models with a high degree of parallelism and complex task dependencies) is not possible, this method cannot be applied in practice.

2. **Accurate modeling using bounding methods for model evaluation**
   Bounding methods tolerate largeness because models are solved using series–parallel reduction instead of creating a state space [18]. Therefore, these methods appear well–suited for scalability analysis. In PEPP, three different bounding methods are implemented in order to select the best bound [5, 16, 11]. Depending on the structure of the graph model, one or the other method will yield the best result. In [6] we have shown for various graph structures that the bounding methods implemented in PEPP are very accurate.

   The applicability of the bounding methods is, theoretically, not limited by the model size, since the time to solution increases only linearly (with the exception of the method of Kleinöder). Typically, the computation of the bounds takes a few seconds to a few minutes for graph models with up to 1000 nodes. But, creating exact models for scalability analysis requires significant memory resources, and these large and complex models are hard to produce and to comprehend.

3. **Approximate modeling**
   For the scalability analysis of a highly parallel program, a large number of graph models must be created

and evaluated. Here, even bounding methods are too time consuming. Therefore, the use of approximate modeling seems to be the best solution. As shown in §3, series–parallel reducible models can be created to compute bounds of the mean runtime. The creation of these models is influenced by the bounding methods presented in §2.2.

Considering these three different methods, we see that approximate modeling is the only viable approach for analyzing the performance scalability of parallel programs using stochastic models. We have shown in [12] that results obtained by evaluating approximate models differ only slightly from the results obtained from evaluating exact models with bounding methods implemented in PEPP. In the following, all results are obtained by evaluating approximate models. In the presented example, we scale the problem size n (i.e. the number of columns) as well as the underlying parallel system. The maximal number of processors, P, working on the problem is limited by the problem size n ($P = n$).

The mean runtimes for 2, 4, and 8 iterations on a 50×40–grid ($n = 50$) are shown in Table 1. It can be seen that the mean runtime is bounded tightly. Only for 8 iterations and increasing degrees of parallelism does the difference between lower and upper bound increase in relative size. In the worst case, the upper bound is 20% higher than the lower bound.

P	2 iterations		4 iterations		8 iterations	
	lower	upper	lower	upper	lower	upper
5	83.28	84.67	164.6	169.3	326.4	338.7
10	43.11	44.45	84.37	88.9	166.1	177.8
15	34.09	34.98	66.94	69.96	132.1	139.9
20	26.43	27.45	51.41	54.94	100.7	109.9
25	18.58	19.71	35.6	39.43	69.05	78.86
30	18.44	19.51	35.42	39.03	68.79	78.06
35	18.26	19.25	35.17	38.5	68.44	77
40	17.99	18.85	34.79	37.71	67.92	75.43
45	17.49	18.12	34.09	36.25	66.94	72.5
50	10.15	11.12	19.13	22.24	36.36	44.49

Table 1: Scalability analysis results for a 50×40–grid

Based on upper and lower bounds, speedup values can be calculated. An upper bound of the mean runtime is a lower bound of the speedup (i.e., this is a value which is reached in any case). The lower bound of the mean execution time defines the value of the speedup which cannot be exceeded. Hence, the upper bound of the mean execution time is more important than the lower bound, since this bound is used to show the reached speedup. In Figure 6 upper and lower bounds for the mean runtime (left) and for the speedup values (right) are shown.

It can be observed that the speedup values increase for parallel systems up to 25 processors. From 30 to 45 processors the speedup values remain nearly constant. The reason is poor load balancing — the 50 columns of the matrix do not spread evenly across these numbers of processors. To analyze the influence of load balancing on the speedup the distribution of the workload among the processors must be considered. Table 2 shows for each configuration of the parallel system how the columns of the grids are distributed

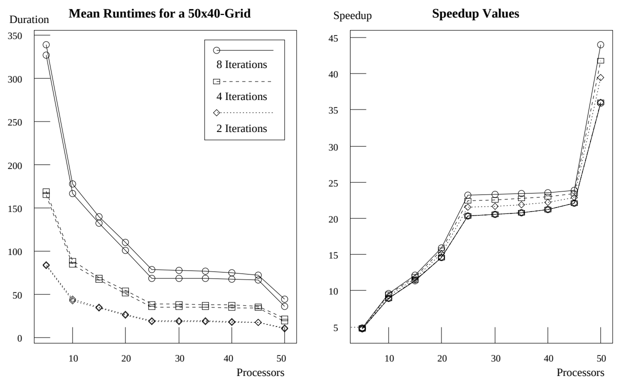


Figure 6: Scalability analysis results for a 50×40–grid

among the processors. Obviously, if all processors have to work on the same number of columns, the speedup is very good. For $n = 50$, the configurations with 5, 10, 25, and 50 processors lead to the best execution efficiencies.

	Problem size n		
p	50	100	200
5	$5 \cdot 10$	$5 \cdot 20$	$5 \cdot 40$
10	$10 \cdot 5$	$10 \cdot 10$	$10 \cdot 20$
15	$10 \cdot 3 + 5 \cdot 4$	$5 \cdot 6 + 10 \cdot 7$	$5 \cdot 14 + 10 \cdot 13$
20	$10 \cdot 2 + 10 \cdot 3$	$20 \cdot 5$	$20 \cdot 10$
25	$25 \cdot 2$	$25 \cdot 4$	$25 \cdot 8$
30	$10 \cdot 1 + 20 \cdot 2$	$10 \cdot 4 + 20 \cdot 3$	$20 \cdot 7 + 10 \cdot 6$
35	$20 \cdot 1 + 15 \cdot 2$	$30 \cdot 3 + 5 \cdot 2$	$25 \cdot 6 + 10 \cdot 5$
40	$30 \cdot 1 + 10 \cdot 2$	$20 \cdot 3 + 20 \cdot 2$	$40 \cdot 5$
45	$40 \cdot 1 + 5 \cdot 2$	$10 \cdot 3 + 35 \cdot 2$	$20 \cdot 5 + 25 \cdot 4$
50	$50 \cdot 1$	$50 \cdot 2$	$50 \cdot 4$
95	—	$5 \cdot 2 + 90 \cdot 1$	$10 \cdot 3 + 85 \cdot 2$
100	—	$100 \cdot 1$	$100 \cdot 2$

Table 2: Grid partitioning for different problem sizes

To calculate how the workload is distributed among the processors of the parallel system, the following formula was used. If n is the number of columns and p the number of processors, then assign

$$\left\lceil \frac{n}{p} \right\rceil \quad \text{columns to} \quad n - \left\lfloor \frac{n}{p} \right\rfloor \cdot p \quad \text{processors and}$$

$$\left\lfloor \frac{n}{p} \right\rfloor \quad \text{columns to} \quad p - \left(n - \left\lfloor \frac{n}{p} \right\rfloor \cdot p \right) \quad \text{processors.}$$

When comparing different problem sizes (Figure 7), we also see that load balancing again has a great influence on the speedup values. Speedups are higher for larger problems, since here, the computation time between synchronization times is higher. Obtained efficiencies are also depicted in Figure 7 (right). It can be seen that also for a good load balancing (e.g. $p \in \{5, 10, 20, 25, 50\}$ for $n = 100$) the efficiency decreases; for $n = 50$ the efficiency goes down to 0.72.

Communication delays have a great impact on performance scalability. Our analysis methods can take these

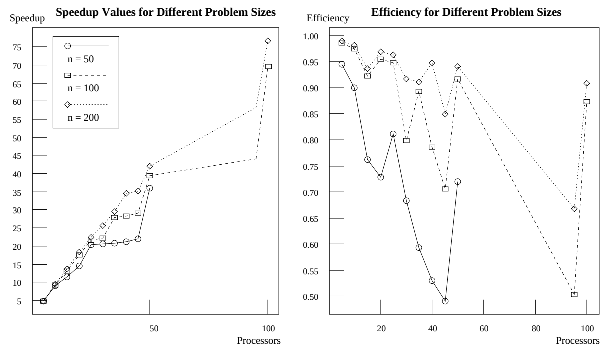


Figure 7: Comparing speedups and efficiencies for different problem sizes

delays into account. In Figure 8, it is shown for neighbor synchronization how approximate models representing communication delays between all processors can be created using the bounding method of Kleinöder (inserting arcs to obtain an upper bound):

1. In the first step, the dependency arcs are shifted to coincide with iteration boundaries (e.g. the arc 3 $\rightarrow$ 2 is moved to 4 $\rightarrow$ 1).
2. Arcs are inserted, so that the communication delays between two processors can be serially reduced resulting in $p - 1$ communication nodes with each node representing 2 communications.
3. All communication nodes are reduced using parallel reduction. This results in a full barrier representing communication delay (2^{p-1} communications). The remaining graph is series–parallel reducible.

Using the method described above, different communication delays can now be considered. For analyzing different communication overheads, the ratio between communication delay and computation duration is varied from $1/10 \ldots 1$. In Figure 9, it can be seen that the speedup is significantly reduced when inserting communication delays. For $n = 50$ speedup goes down from 36 without communication to 13 with communication delays equal to the computation duration (ratio=1).

The diagrams presented in Figures 6–9 verify the necessity of a systematic scalability analysis. To obtain these results with measurements, more than 300 measurements must be taken.

Approximate models analyzing the influence of communication can also be created using the method of Dodin. Here, a model for calculating an upper bound is created by duplicating nodes. In Figure 10 the creation of such models for neighbor synchronization is shown. The nodes of one iteration are duplicated (step 1) in order to allow series–parallel reduction (step 2).

When solving models for neighbor synchronization with this method, a formula can be given to obtain the upper bound. Let T_i denote the execution time distribution of all tasks in column i of the model. Further, let C be the

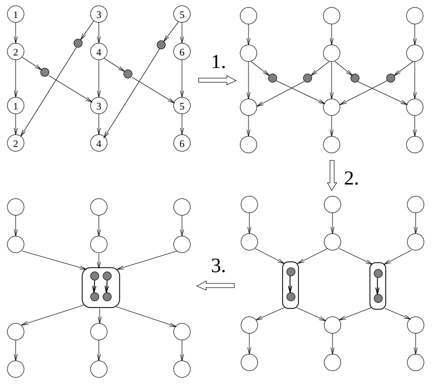


Figure 8: Modeling the influence of communication (method of Kleinöder)

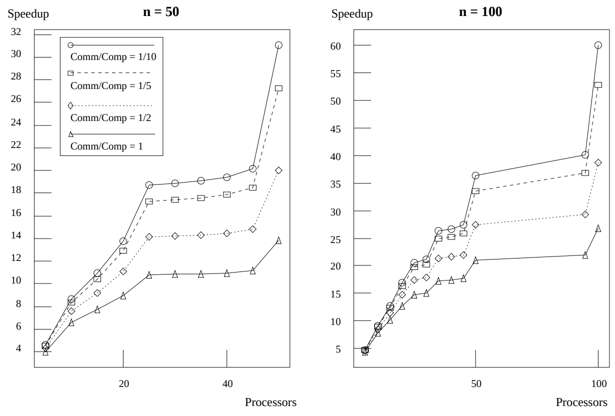


Figure 9: Influence of communication on the performance scalability

communication time distribution and p the number of processors. Then, the upper bound T_u for a neighbor structure with communication times can be computed recursively as follows (see Figure 10):

$$
\begin{aligned}
T_i^0 &= T_i \qquad (\forall\, i)\,(1 \le i \le p) \\
T_i^{k+1} &= \begin{cases} T_i^0 + \max(T_i^k,\ (T_{i+1}^k + C)) & (i = 1) \\ T_i^0 + \max(T_i^k,\ (T_{i-1}^k + C) & (i = p) \\ T_i^0 + \max(T_i^k,\ (T_{i-1}^k + C), (T_{i+1}^k + C)) & \\ & (1 < i < p) \end{cases} \\
T_u &= \max\left\{ T_i^{h-1} \;\mid\; i = 1, \ldots, p \right. \\
&\qquad \left. \wedge\;\; h = \text{height of the neighbor structure} \right\}
\end{aligned}
$$

T_i^{k+1} denotes the upper bound for the completion time of the i-th node on iteration level k.

We have verified for small examples that bounds obtained with this methods are better than bounds obtained with the method of Kleinöder. This agrees with cases where there was a high degree of parallelism [6]. For evaluating large models, a tool is needed to compute the bounds. With the recursive formula presented above, this tool only has

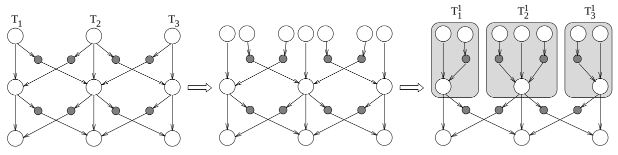


Figure 10: Modeling the influence of communication (method of Dodin)

to work with numerical distributions; the graph model is actually not needed.

5 Conclusion and Prospect

Scalability analysis is an important issue when implementing parallel programs for scaled parallel systems. A systematic approach must be established wherein scaled performance can be estimated subject to the constraints of the analysis tool used. In this paper, we have presented an approach for scalability analysis based on stochastic graph modeling. There are several compelling reasons for a model–based approach from a performance evaluation standpoint, but the solution techniques must be efficient in order to return results in a timely manner.

We have verified that scaled models can be created from a generic computation description in PDL and analyzed by our stochastic graph analysis tool, PEPP. Our results indicate that scalability analysis is possible with this approach and delivers performance predictions that are consistent with other solution techniques.

However, there are still many open issues to address. We have only briefly touched how the machine model interacts with the analysis. We are currently exploring this issue more thoroughly through the analysis of additional testcases. However, one benefit of stochastic graph modeling in this regard is its support of model composition. That is, a library of analyzed submodels can be developed and these components can be used as building blocks for more complex models; solutions for submodel components can be plugged in during the analysis of the larger model. Finally, we are investigating the integration of scalability model generation into PEPP. We believe that the model-based instrumentation support in PEPP may allow us to extrapolate a template of a generic model of programs from measurements of a few of its scaled versions. This appears particularly important when task density functions are unknown.

References

[1] V. S. Adve and M. K. Vernon. The Influence of Random Delays on Parallel Execution Times. *Perf. Eval. Review*, 21(1):61–73, 1993.

[2] M. Ajmone Marsan, G. Balbo, and G. Conte. A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems. *ACM Trans. on Comp. Sys.*, 2(2):93–122, May 1984.

[3] F. Bodin, P. Beckman, D. Gannon, S. Yang, A. Malony, and B. Mohr. Implementing a Parallel C++ Runtime System for Scalable Parallel Systems. In *Proc. SC '93*, 1993.

[4] H. Burkhart and R. Millen. Performance Measurement Tools in a Multiprocessor Environment. *IEEE Trans. on Comp.*, 38(5):725–737, May 1989.

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

[6] F. Hartleb and V. Mertsiotakis. Bounds for the Mean Runtime of Parallel Programs. In R. Pooley and J. Hillston, Eds., *Sixth Int'l. Conf. on Model. Tech. and Tools for Comp. Perf. Eval*, pp. 197–210, Edinburgh, 1992.

[7] F. Hartleb. Stochastic Graph Models for Performance Evaluation of Parallel Programs and the Evaluation Tool *PEPP*. TR 3/93, Universität Erlangen-Nürnberg, IMMD VII, 1993.

[8] U. Herzog. Formal Description, Time and Performance Analysis. In T. Härder, H. Wedekind, and G. Zimmermann, Eds., *Entwurf und Betrieb Verteilter Systeme*, Berlin, 1990. Springer Verlag, Berlin, IFB 264.

[9] U. Herzog and W. Hofmann. Synchronization Problems in Hierarchically Organized Multiprozessor Computer Systems. In M. Arato, A. Butrimenko, and E. Gelenbe, Eds., *Performance of Computer Systems – Proceedings of the 4th Int'l. Symp. on Model. and Perf. Eval. of Comp. Sys.*, Vienna, Austria, Feb., 6–8 1979.

[10] H. Jonkers. Queueing Models of Parallel Applications: The Glamis Methodology. *Proc. of the 7th Int. Conf. on Model. Techn. and Tools for Comp. Perf. Eval.*, 1994.

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

[12] A.D. Malony, V. Mertsiotakis, and A. Quick. Automatic Scalability Analysis of Parallel Programs Based on Modeling Techniques. *Proc. of the 7th Int. Conf. on Model. Techn. and Tools for Comp. Perf. Eval.*, 1994.

[13] A. Quick. A New Approach to Behavior Analysis of Parallel Programs Based on Monitoring. In G.R. Joubert, D. Trystram, and F.J. Peters, Eds., *ParCo '93: Conf. on Parallel Computing, Proc. of the Int. Conf., Grenoble, France, 7–10 Sept. 1993*. Advances in Parallel Computing, North–Holland, 1993.

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

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

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

[17] F. Sötz. A Method for Performance Prediction of Parallel Programs. In H. Burkhart, editor, *CONPAR 90–VAPP IV, Joint Int'l. Conf. on Vector and Parallel Processing. Proc.*, pp. 98–107, Zürich, Switzerland, Sept. 1990. Springer–Verlag, Berlin, LNCS 457.

[18] K.S. Trivedi and M. Malhotra. Reliability and Performability Techniques and Tools: A Survey. In B. Walke and O. Spaniol, Eds., *Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen*, pp. 27–48, Aachen, Sept. 1993. Springer.

[19] H. Wabnig, G. Kotsis, and G. Haring. Performance Prediction of Parallel Programs. In B. Walke and O. Spaniol, Eds., *Proc. der 7. GI–ITG Fachtagung "Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen", Aachen, 21.–23. Sept. 1993*, pp. 64–76. Informatik Aktuell, Springer, 1993.

[20] M. Wolfe. High Performance Compilers. Monograph, Oregon Graduate Institute, 1992.

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