published in:
7th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation,
May 4 – 6, 1994, Vienna, Austria

Automatic Scalability Analysis of Parallel Programs Based on Modeling Techniques

Allen D. Malony[1], Vassilis Mertsiotakis[2], Andreas Quick[3]

[1] Dept. of Computer and Information Science, University of Oregon, Eugene, OR 97403, USA
[2] University of Erlangen-Nürnberg, IMMD VII, Martensstr. 3, 91058 Erlangen, Germany
[3] Thermo Instruments Systems, Frauenauracher Str. 96, 91056 Erlangen, Germany
malony@cs.uoregon.edu, {vsmertsi,quick}@informatik.uni-erlangen.de

Abstract

When implementing parallel programs for parallel computer systems the performance scalability of these programs should be tested and analyzed on different computer configurations and problem sizes. Since a complete scalability analysis is too time consuming and is limited to only existing systems, extensions of modeling approaches can be considered for analyzing the behavior of parallel programs under different problem and system scenarios.

In this paper, a method for automatic scalability analysis using modeling is presented. Initially, we identify the important problems that arise when attempting to apply modeling techniques to scalability analysis. Based on this study, we define the *P*arallelization *D*escription *L*anguage (PDL) that is used to describe parallel execution attributes of a generic program workload. Based on a parallelization description, stochastic models like graph models or Petri net models can be automatically generated from a generic model to analyze performance for scaled parallel systems as well as scaled input data.

The complexity of the graph models produced depends significantly on the type of parallel computation described. We present several computation classes where tractable graph models can be generated and then compare the results of these automatically scaled models with their exact solutions using the PEPP modeling tool.

1 Introduction

Implementing parallel programs for scalable parallel systems is difficult since the program's behavior could vary for different problem sizes and different system configurations. In order to implement portable and efficient programs which will also have good performance scalability, parallelization choices must be tested for many systems and problem testcases. Such empirical analysis is time consuming and is limited to existing parallel computer systems.

Modeling parallel programs with discrete event models like stochastic graph models [15] or stochastic Petri nets [1] is a well–known and proven method to analyze a program's dynamic behavior. It can be used to predict the program's execution time [16], 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.

When using modeling for scalability analysis, we desire to compute speedup values from the predicted runtimes of model instances for different numbers of processors or problem sizes. As done with performance monitoring, we also want to use modeling to analyze different parts of the program in order to obtain a detailed scalability profile [2, 11]. A significant advantage of modeling vs. monitoring is that model–based analysis is not restricted to existing systems and does not, necessarily, require access to existing systems for experimentation. Thus, we hope to be able, using modeling, to evaluate whether it is worth scaling a parallel machine and what the best scale of the system would be.

A systematic scalability analysis based on modeling techniques requires the creation of models for different configuration and topologies of the parallel system as well as for different problem sizes. However, model creation is a difficult problem – issues such as problem mapping and processor scheduling can quickly lead to large model complexity. One approach for automatic analysis 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. Adopting this idea, we have developed the *P*arallelization *D*escription *L*anguage (PDL) for describing the structure of parallel programs, the parallelization scheme for each parallel program part, and various aspects of a program's runtime behavior.

Although this "generator" approach could be undertaken with different modeling techniques, we first target stochastic graph modeling for scalability analysis and automatic model generation. In several respects, scalability analysis using stochastic graph models is the most challenging one because of the close association between model complexity and solution tractability and accuracy. The key issue 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. To evaluate the efficacy of our techniques, we have integrated methods for model generation and scalability analysis into our tool PEPP (*P*erformance *E*valuation of *P*arallel *P*rograms) [4]. Based on the parallelization description language, PDL, a model generator for other model targets (like stochastic Petri net models) can be implemented in a similar manner.

The remainder of the paper is organized as follows. In section 2, the concept of model–based scalability analysis is introduced. The automatic creation of scalability models is addressed in section 3. Here, different parallel computation classes are described, and generic and scaled models of those classes are discussed. Our parallelization description language is presented in section 4. Finally, in section 5 it is shown how scalability analysis with stochastic graph models using our modeling tool PEPP can be carried out.

2 The Methodology

Modeling programs to be executed on parallel or distributed systems is too complicated a task to develop a model from scratch. For this reason Herzog proposed a "three step methodology" [9] to reduce modeling complexity. 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).

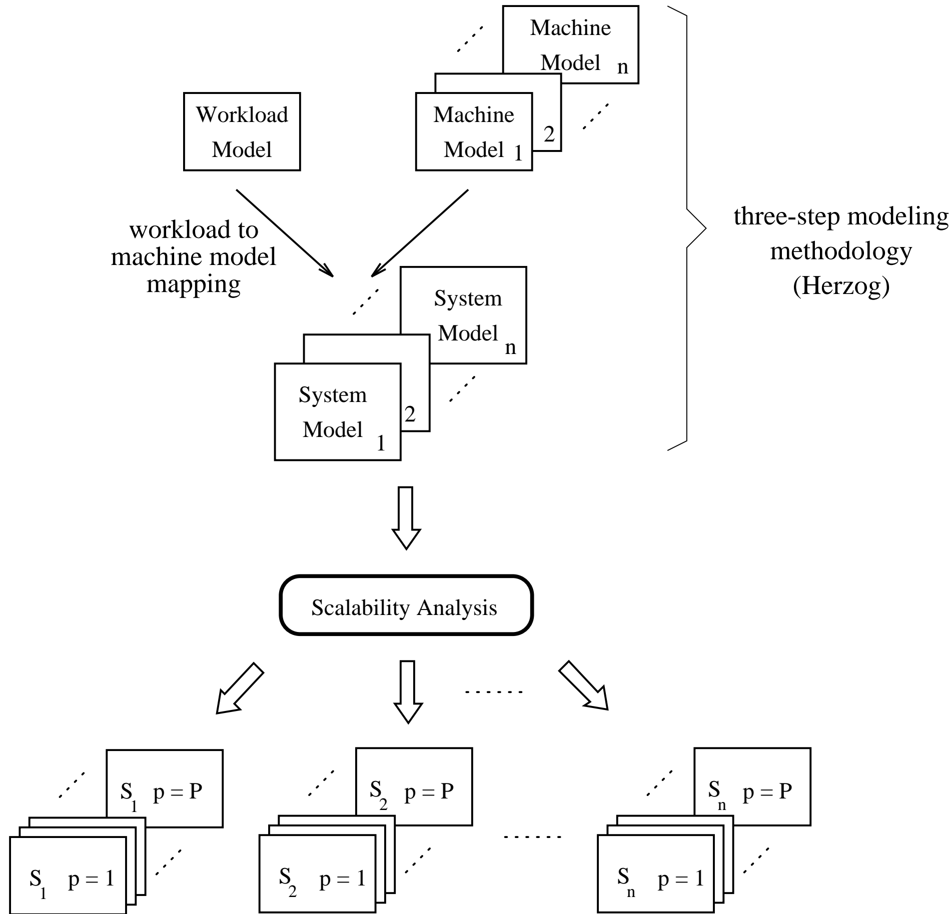


Figure 1: Concept for Automatic Scalability Analysis

Our approach for scalability analysis uses this methodology as a basis, but extends it by another necessary step: a system model is enumerated to model various degrees of parallelism varying from $p = 1$ to $p = P$, where P is the maximum number of processors considered for scalability analysis. Since the number of processors of the parallel or distributed machine is not fixed in scalability analysis, the number of processors should not — in contrast to Herzog's approach — be set in the machine model. The mapping of the workload model onto the machine model must be realized for each number of processors separately by creating (ideally, automatically) *scaled models*. In our approach, the machine model only describes synchronization mechanisms and performance distributions for each important machine component.

Unfortunately, techniques that automatically generate scaled models must address several difficult issues. 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. The difficulty, however, is that some parallelization schemes, although simple, can significantly impact scaled graph model complexity — resulting in solution intractability — if exact execution behavior is modeled. Although modeling techniques have been developed that are "largeness tolerant" [17] (i.e., can deal to some extent with graph complexity), the process of creating a correct and accurate graph model is non–trivial. In order to overcome these model generation and evaluation problems, we should instead develop approximate models which can capture the

correct behavior of the program, but that do not sacrifice analysis accuracy. Approximate model generation, however, is not easy due to problems such as task dependencies, scheduling, and synchronization. Furthermore, the generation of task density functions for approximate scaled models presupposes intimate knowledge of performance interactions between parallel tasks. Techniques must be developed for both graph structure approximations and execution time distribution approximations, keeping in mind, of course, that naive approximations can lead to invalid analysis results.

Finally, it is our aim to implement not only scalable, but also portable parallel programs. Therefore, our work should also address the use of different machine models 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. Using this methodology, model–based scalability can be used to compare the scalability performance of different machines.

3 Scalability Models for Parallel Programs

Using stochastic graph models, the execution order of program activities, their runtime distribution, and branching probabilities can be represented. 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 dependences 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).

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. The principal problem to solve is representational. That is, how scalability properties of a program – which might be known at different levels of detail and accuracy – are represented in a manner that a modeling tool can use.

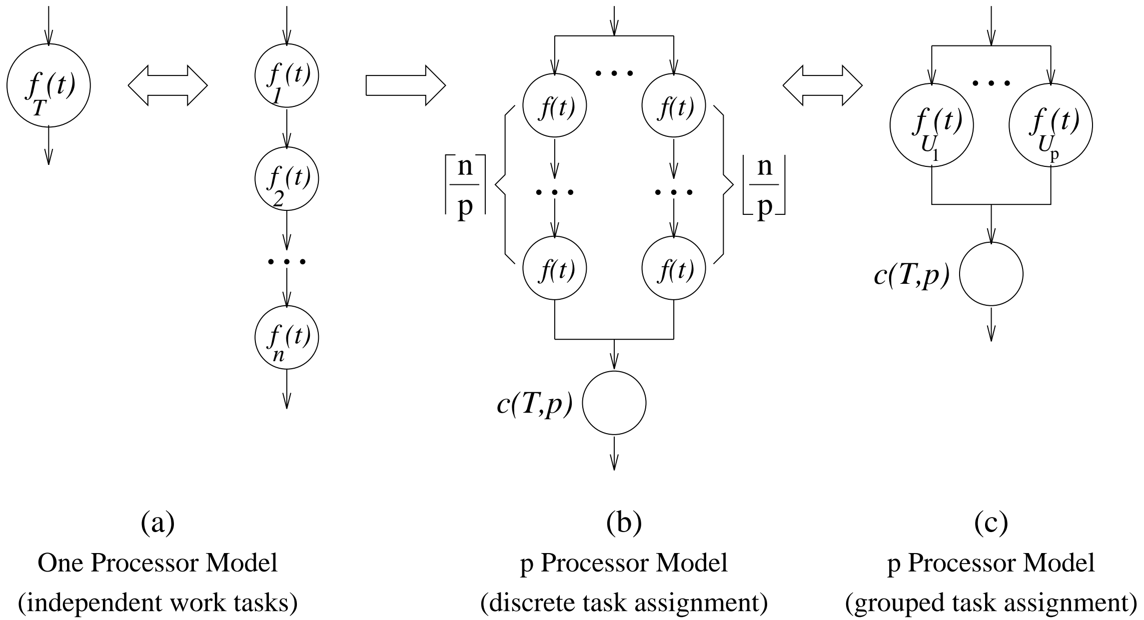
We believe that this problem is best approached by considering different computation classes. In this section we consider several computation classes that are related to well-known parallel execution paradigms. We attempt to define the scaling behavior of these classes and to formulate how scaled models will be developed for them.

3.1 Parallelization of Independent Tasks

Perhaps the simplest parallel execution paradigm is one of n equivalent tasks executing on p processors. If $n > p$, then the tasks must be assigned to the processors. A trivial form of assignment is a dynamic one where each processor takes a task, executes it, and retrieves another until all tasks are completed. Given one processor, the tasks execute sequentially. Given $p \leq P$ processors, p tasks can be executing in parallel.

Let v_i be a task in the set of n tasks, V, that are to be executed. Suppose $f_i(t)$ represents the density function of the execution time of task v_i, for $1 \leq i \leq n$. Under the assumption of identically distributed tasks, $f_i(t) = f_j(t), \forall i, j \leq n$, let $f_T(t)$ represent the overall distribution density. In this case, the one processor (i.e., sequential) execution time is given by the convolution of all

densities $f_i(t)$ resulting in the density $f_T(t) = \oplus_{i=1}^{n} f_i(t)$. The graph models in Figure 2(a) show two graph model versions of the sequential case.



(a)
One Processor Model
(independent work tasks)

(b)
p Processor Model
(discrete task assignment)

(c)
p Processor Model
(grouped task assignment)

Figure 2: Independent, Identically Distributed Tasks

From the single processor graph description, what is needed to allow a scaled analysis? In this case, very little. We know that the tasks must be assigned to at least one processor. Although it will be critically important to know the scheduling strategy in later cases, here only one strategy makes sense because the tasks are identically distributed. Knowing this, the modeling tool can easily generate the graph model shown in Figure 2(b), where the tasks are assigned to independent paths (representing processors) as equally as possible; each path has a subset of tasks, represented by U_1, ..., U_p. The tool can then reduce this graph to the simpler one shown in Figure 2(c). The important thing to note is that separate paths have the same mean execution time only when n mod $p = 0$. In any case, the mean completion time of the task set V will be determined by all subsets.

Even with this simple model, there are several issues to address. First, even though tasks are independent, it is often the case that their parallel execution can influence their execution time, due to resource contention in the target system. This contention will be a consequence of the machine model and the workload mapping. If we chose to model low-level contention overhead due to parallel interactions, the scaled models may become too complex to solve, even in this simple case. Another approach would be to model the accumulated delay caused by the interactions using a single distribution function based on the type(s) of interactions expected, the number of tasks, and the number of processors. With respect to the graph model, this would require an "interaction" or "contention" node to be added; this node is shown in Figure 2(b,c)) with the distribution function $c(T, p)$.

If the tasks are independent, but not identically distributed, $f_i(t) \neq f_j(t)$, performance will depend significantly on task load balance. Several static scheduling scenarios could be applied with accumulative "per processor" distributions computed by the convolution of the densities of tasks assigned to a processor, $f_{U_i}(t) = \oplus f_j(t), v_j \in U_i$. A dynamic scheduling scenario is more

difficult to model, but a simple lower bound estimate is given by $\frac{\oplus f_j(t)}{p}$. As in the identically distributed case, a contention node is added at the end.

3.2 Parallelization of Dependent Tasks

Of course, any parallel program will have portions of the computation where the executing tasks are inter–dependent. The resulting models have a more complex structure than the models discussed in section 3.1 due to the synchronization arcs in the task graph. In general, dependencies can be quite arbitrary. In practice, certain dependent parallel computation classes are quite common in real–world applications. In the following, we discuss model scalability for some standard computation classes; Figure 3 shows some of the classes we will be considering.

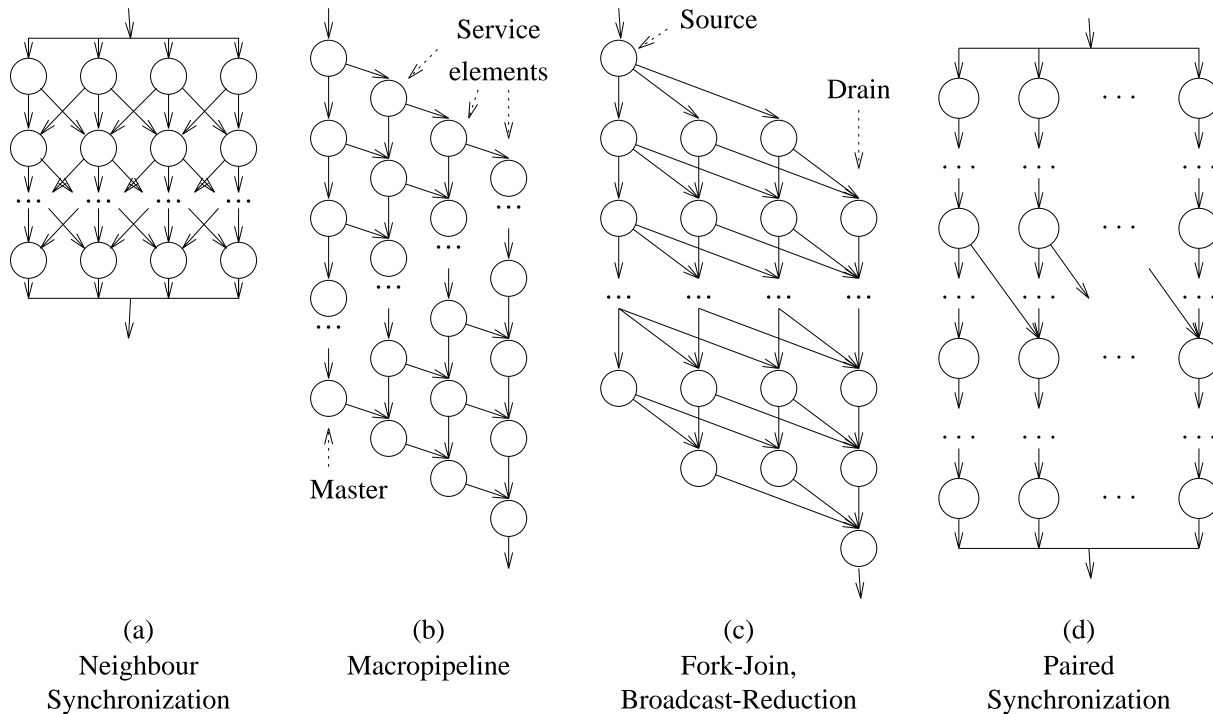


Figure 3: Parallel Computation Classes with Dependent Tasks

Before treating specific cases, 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 computation, to a tractable graph model.

Structurally, the scaled model should be of a form that tools like PEPP can analyze. The number of task nodes and the dependency structure of even basic computation graphs can overwhelm graph modeling tools. 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. Also, we want to use graph structures that can be reduced during model analysis. However, the trick will be to perform model scaling in a way that does not sacrifice modeling accuracy. 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.

3.2.1 Neighbor Synchronization

Many iterative solution methods for linear equation systems can be modeled by a neighbor structure (Figure 3(a)). The main characteristic of this computation class is that a processor can start the i-th iteration only after the $(i-1)$-th iteration has finished on its neighbor processors [10]. (Although Figure 3(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 4. If we were to represent, in the scaled model, each task and dependency in the generic model, the graph size and complexity would be unmanageable. However, it is easy to identify that the tasks at each iteration are independent and could be modeled by the techniques in section 3.1, but the neighbor synchronization has to be simplified. Our approach is to collapse the neighbor synchronization between iterations to a single barrier, as shown in Figure 4.

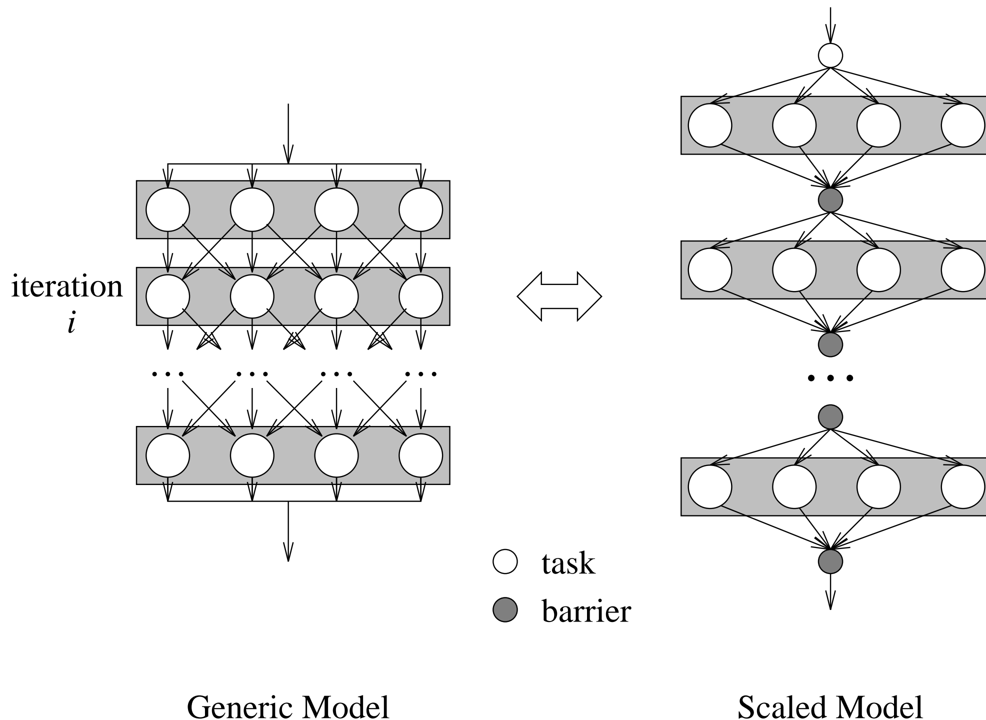


Figure 4: Scaling Neighbor Synchronization Models

Although we can accurately capture the per iteration task execution times in the scaled model (albeit using a scaled independent task model), the synchronization approximation has certain

model analysis consequences. In particular, because synchronization and computation cannot overlap in the scaled model, it will have poorer worst case behavior than expected for an exact model. However, this effect is reduced if the tasks at each iteration are identically distributed, or if the number of tasks at each iteration is significantly larger than the number of processors. The principal advantage of the scaled model is that it can be easily analyzed by graph modeling tools.

3.2.2 Macropipeline

If a task can be divided into subtasks, where the result of one subtask is the input to another subtask, the macropipeline computation paradigm (Figure 3(b)) results. The structure of a macropipeline is characterized by the number of service elements and the number of jobs. The execution time is determined by the interarrival time of new jobs and the runtime distribution of one pipeline stage. The macropipeline task graphs have also been referred to as mesh graphs [21] and are characteristic of wavefront computations.

Our scalability approach is similar to that for neighbor synchronization (Figure 5). We identify sets of independent tasks and separate their parallel execution in the approximate graph model by barrier synchronization. Many of the analysis ramifications are also the same. The graph structure is more complex, however, in that a different number of tasks are present at each graph stage. It is also possible that the tasks will be of different types, complicating the scaled submodel for independent tasks.

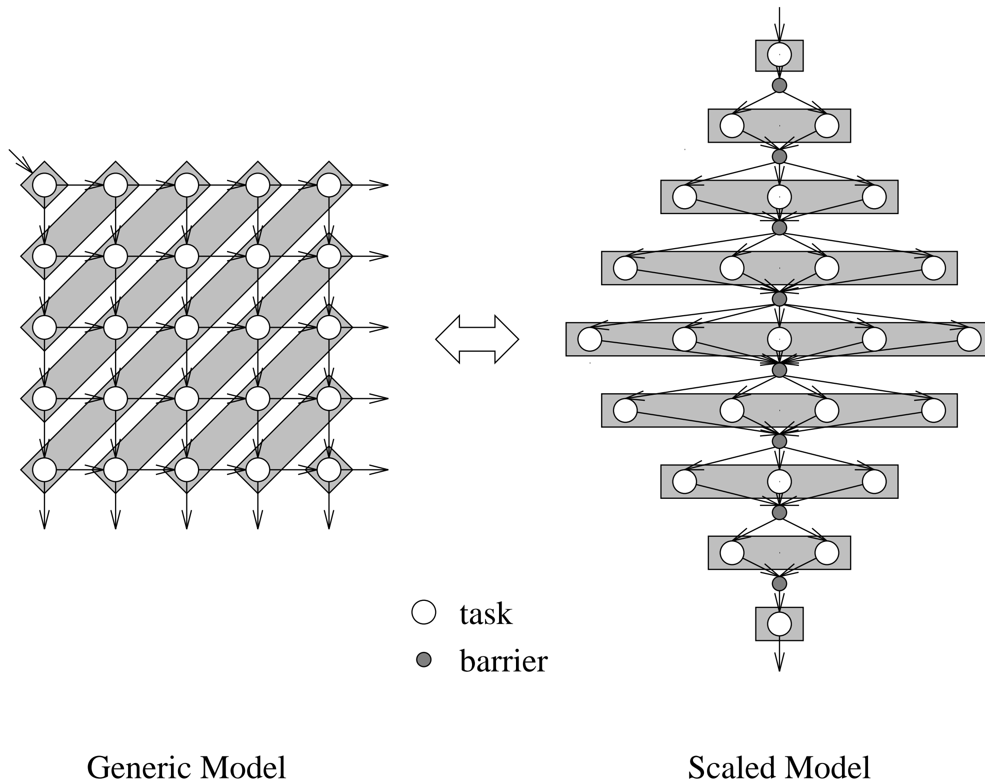


Figure 5: Scaling Macropipeline Models

3.2.3 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 3(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 6; 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 a simpler scaled model. 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 model is less likely to lead to modeling inaccuracies.

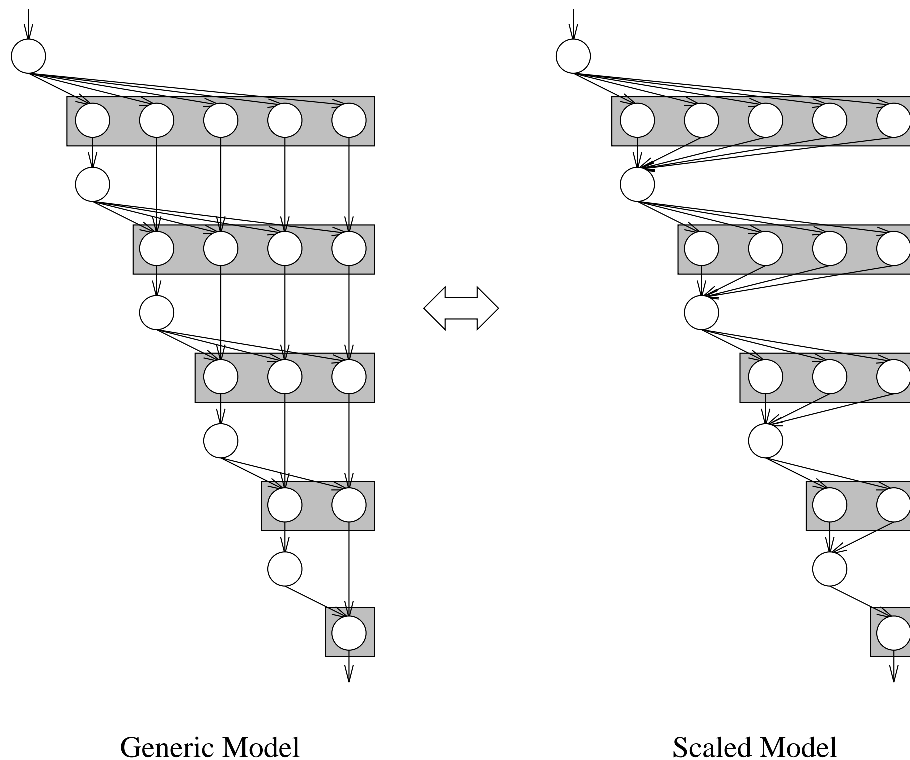


Figure 6: Scaling Fork-Join, Broadcast-Reduction Models

3.2.4 Paired Synchronization

Our last computation class is commonly found in parallel loops with static dependencies [20]. In general, we are considering parallelizable loop statements, where the loop body consists essentially of independent and dependent parts. The independent parts can be executed at any time. The dependent parts, however, have to wait for the results of the corresponding independent part and several earlier iteration steps. Often, a single static dependency spans two loop iterations separated by a constant number of iterations, resulting in a paired synchronization between two task nodes, as in Figure 7. However, in general, the number of static dependencies between loop iterations can be greater than two, as in the following loop:

```
do i = 1, n
   ...
   a[i] = a [i-1] + a[i-5] + a[i-11]
   ...
done
```

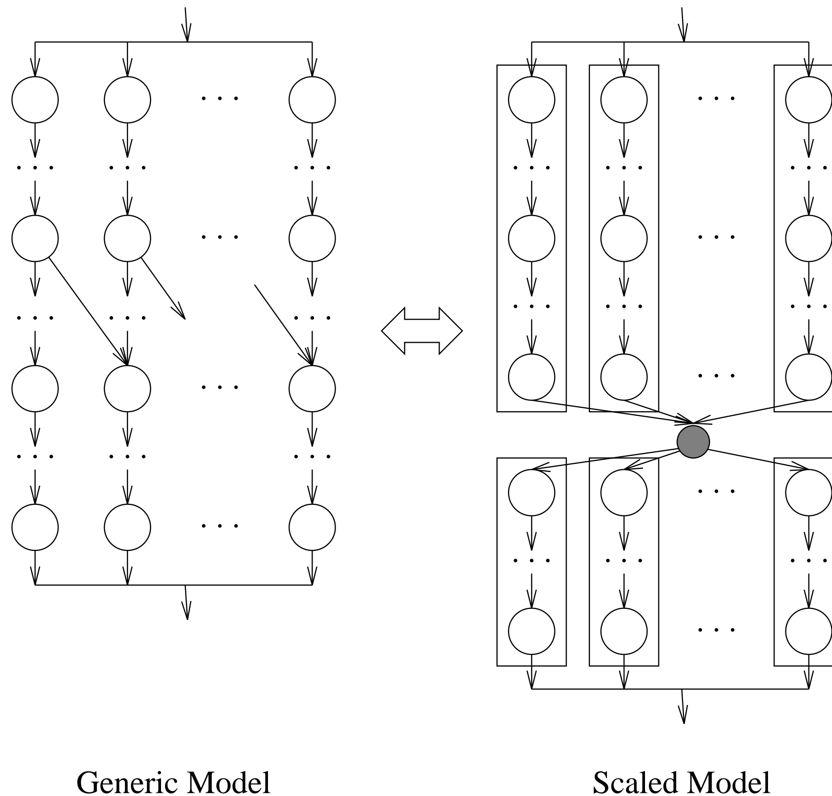


Figure 7: Scaling Paired Synchronization Models

Because the dependency constraints effectively separates an iteration into independent and dependent parts, we can scale the generic model by forcing a barrier synchronization before entering the dependent task nodes. Notice that we now have an option on how to parallelize the task sections. Shown in the figure is an assignment of processors to loop iterations.

For the scalability analysis of a program that contains independent and dependent working phases as described above, the following parameters are needed: the problem size (i.e. the number of iterations of the particular loop), the runtime distributions of the independent and dependent subtasks, and, finally, the scheduling strategy. The latter is especially important to enable the modeling tool to generate a model for any number of processors.

The presented list of computation classes is not complete. In order to analyze more general structures, a default parallelization scheme is provided, where the problem structure must be given in terms of a directed acyclic graph.

To conclude this section, the presented scaled models were gained from our experience in model evaluation using various bounding methods, since most of them apply modifications to the models until they are series–parallel reducible. In [13] different bounding methods are compared in order to obtain good scaled models.

4 The Parallelization Description Language PDL

In order to carry out scalability analysis of a parallel program based on modeling techniques, a general description of the program to be analyzed is needed, describing the different workload phases of the program's computation (workload model). We define the *P*arallelization *D*escription *L*anguage (PDL) for describing how a program can be parallelized. We consider a program to consist of an arbitrary pattern of execution phases of the following types: sequential phase, parallelizable phase, and synchronization phase. Following the methodology presented in section 2, a workload model described in this manner can be combined with a machine model to build a system model, forming the basis for scalability analysis. The choice of a specific parallel architecture mainly influences a subtask's runtime, the mechanism and the duration of barrier synchronizations, and the communication times.

After creating the system model the user has to select two parameters needed for scalability analysis: the problem size and the maximal degree of parallelism, P (i.e., the maximum number of processors to be considered with this analysis). With these two input parameters, P system

models are created automatically to carry out scalability analysis (Figure 8). The scaled models are derived using the techniques discussed in the previous section. (Note, if P is large, only certain numbers of processors may be selected for scalability study.)

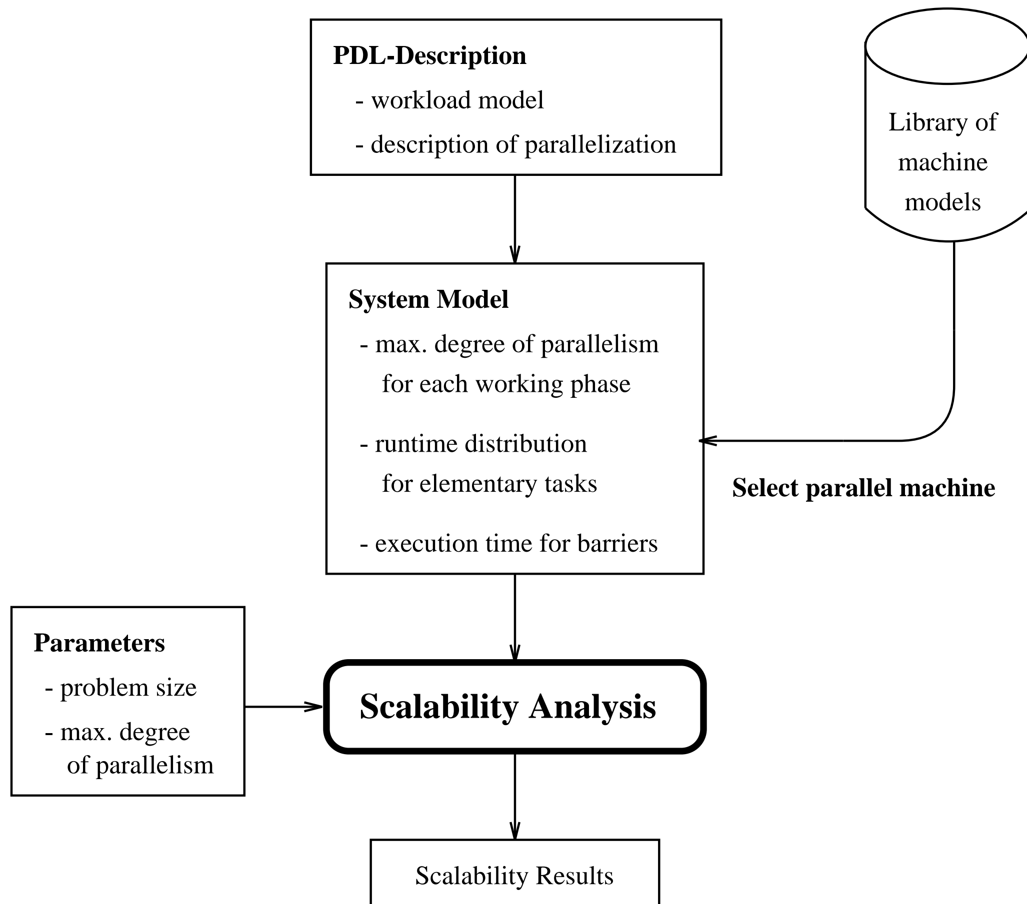


Figure 8: Scalability Analysis Using PDL

4.1 Program Description

A description of a parallel program in PDL consists of the following three parts:

1. **Execution pattern**
   The execution order of the different work phases and synchronization phases must be specified in terms of a regular expression.

2. **Definition of the work phases**
   As the work phases represent the parallelizable parts of the program, the runtime distributions of the subtasks and the parallelization scheme for a particular work phase have to be specified.

3. **Concatenation of work phases**
   If no synchronization phase is between two parallelized work phases, they may be concatenated in different ways. The selected concatenation method must be described for each

```
PROGRAM NAME IS 'prgA';
EXECUTION PATTERN IS w1-> sync-> w2-> w2-> w3;
```

Figure 9: Begin of a PDL–Description

work phase which is not succeeded by a synchronization phase or a sequential phase (see section 4.3).

The specification of the program's name and the execution pattern form the first part of each PDL description. The name of the described program must be given for reference purposes; see Figure 9. After this, the work phases are described.

4.2 Description of Work Tasks

In general, we distinguish between two types of work phases, namely work phases with a predefined parallelization scheme and work phases with an arbitrary parallelization scheme; see section 3.2. The definition of a work phase with a predefined parallelization scheme consists of the following elements:

- **Name of the work phase**
  For identifying the work phase in other parts of a PDL description a unique name must be assigned. This name is used for specifying the execution pattern (Figure 9) and the concatenation of two phases (Figure 12).

- **Parallelization scheme**
  Using PDL, the following six parallelization schemes (computation classes) are available (see section 3.2): `INDEPENDENT`, `NEIGHBOR`, `NEIGHBOR-TORUS`[1], `MACROPIPELINE`, `FORK-JOIN`, and `PAIRED`. For some of the parallelization schemes the partitioning of the data and the mapping of the partitions onto the processors must be given.

- **Runtime distributions**
  The user can choose between various parametric distributions like exponential, general Erlangian, or numerical distributions. Since the runtime distribution is a property of each task and independent of the selected machine, the runtime distribution must be specified in PDL. In order to represent the dependence on the underlying machine, a runtime factor (`RTFactor`) can be given. This factor is defined in the machine model and allows the comparison between different parallel systems. The use of numerical distributions which may be obtained from measurements makes the accurate modeling of real–world programs possible. Using parameters obtained from monitoring renders performance prediction more relevant.

[1] The neighbor–torus class is a special case of the neighbor class were the tasks at the left and the right borders are also synchronized.

- **Problem size specification**
  This depends on the selected parallelization scheme. The problem size may be either a constant or a data structure which can be varied and whose size must be given by the user as a parameter. A variable data structure can be an array, a grid, or any other partitionable structure. In the case of a macropipeline or a fork–join model, the number of jobs and their interarrival times have to be specified. In Figure 10, both cases are shown.

- **Communication volume**
  For the generation of accurate system models, the amount of data to be transferred between processors must be known. By specifying the communication volume the underlying architecture can easily be changed. It must be specified in the machine model whether the data exchange is realized via message passing or via shared memory access. The time needed for data exchange is also specified in the machine model. In general, the duration is a function of the communication volume given in the PDL–description. Since the communication volume is dependent on the problem size, it has to be specified as a function of the problem size in case of a variable problem size.

The description of the three work phases from Figure 9, each with a predefined parallelization scheme, is shown in Figure 10 and Figure 11. For the description of work phases which do not match any of the predefined computation classes, PDL provides the parallelization scheme `ARBITRARY` (see section 3.2).

```
WORK PHASE w1 IS:
    PARALLELIZATION SCHEME IS INDEPENDENT;
    SUBTASK RUNTIME IS ERLANG(100,1*RTFactor);

    // constant problem size
    PROBLEM SIZE IS VECTOR A(100);
    COMMUNICATION VOLUME IS 100*SIZEOF(INT);
END;

WORK PHASE w2 IS:
    PARALLELIZATION SCHEME IS NEIGHBOR;
    SUBTASK RUNTIME IS NUMERICAL(''dur_b.dis.$ARCH'');

    // variable problem size
    PROBLEM SIZE IS GRID B(height, width);
    COMMUNICATION VOLUME IS height*SIZEOF(FLOAT);
END;
```

Figure 10: Description of Work Tasks with `INDEPENDENT` and `NEIGHBOR` Parallelization

4.3 Concatenation of Work Phases

As already mentioned, sometimes it is desirable to model the execution of two work phases in series without a global synchronization between them. In this case, the user has to describe how

```
WORK PHASE w3 IS:
   PARALLELIZATION SCHEME IS PAIRED;
   // i-th iteration depends on (i − 1)-th and (i − 5)-th iteration
   DEPENDENCY IS 1, 5;

   INDEPENDENT SUBTASK IS:
      RUNTIME IS ERLANG(5,1*RTFactor);
   END;

   DEPENDENT SUBTASK IS:
      RUNTIME IS DETERMINISTIC(5*RTFactor);
      COMMUNICATION VOLUME IS 1*SIZEOF(FLOAT);
   END;

   PROBLEM SIZE IS GRID C(height, width);
   COMMUNICATION VOLUME IS height*SIZEOF(FLOAT);
END;
```

Figure 11: Description of a Work Task with `PAIRED` Parallelization

to join the work phases. The following options are available: `NEIGHBOR`, `NEIGHBOR-TORUS`, `TORUS`, `SERIAL`, and `BARRIER`. Figure 12 shows how the concatenation of two work phases can be described in PDL.

```
CONCATENATION IS:
   // concatenation w1, w2 must not be specified,
   // since there is a synchronization between both phases
   w2, w2  : NEIGHBOR;
   w2, w3  : SERIAL;
END;
```

Figure 12: Concatenation of Work Tasks

4.4 Flexibility of PDL

For flexible scalability analysis, the selected problem size and the underlying parallel machine should not be determined in the PDL–description. This has the advantage that it is not necessary to change the PDL–description each time the user wants to perform a new scalability analysis for different input data.

Therefore, PDL supports language constructs to describe program parts which are dependent on these input parameters on an abstract level. When creating scaled models the model generator replaces these variables by their actual input values. Such variables can be used to describe the problem size and the execution time of tasks in the PDL description. In Figure 10 the variables `height` and `width` are used to allow a flexible description of the problem size.

The abstract description of the communication volume in the same example (Figure 10) enables the abstraction from a real machine. Here the advantage of our modeling method becomes clear. Depending on the selected machine, the communication volume changes from 400 byte (size of an integer is 4 byte) to 800 byte, if an integer is represented by 8 bytes.

The above discussed examples demonstrate that the language PDL can be used to describe a wide range of parallel structures. The information given for each work phase is sufficient to create stochastic models for performance modeling. Since PDL is not dependent of any model type, a description in PDL can be used to generate arbitrary stochastic models like stochastic graph models or stochastic Petri net models. Since we have developed a modeling tool for efficient evaluation of stochastic graph models, we show in the next section, how a PDL–description can be used in combination with this tool to automatically create scaled stochastic graph models.

5 Automatic Scalability Analysis with the Tool PEPP

PEPP (*P*erformance *E*valuation of *P*arallel *P*rograms) [4] is a modeling tool for analyzing stochastic graph models. It provides various evaluation methods to compute the mean runtime of a modeled program. PEPP supports the analysis of real–world applications by applying efficient solutions methods including a series–parallel structure solver, an approximate state space analysis, and bounding methods to obtain upper and lower bounds of the mean runtime [8]. In order to model measured runtimes, numerical runtime distributions are allowed in all three cases. The following example illustrates how solutions to large graph models are calculated using bounding methods.

For systematic monitoring of parallel and distributed programs, PEPP implements the M^2–cycle methodology [14]. Here, a functional model of a program to be measured is used for event selection, automatic program instrumentation, and event trace evaluation. allowing the functional model to be extended into a performance model.

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

As done with performance monitoring, modeling can classify different parts of the program in order to obtain a detailed scalability profile (loss analysis [3, 2]). The relative influence of the different program phases on the program's execution time can be determined in the model. For this, 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.

5.1 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. This example is influenced from an application running on the multiprocessor SUPRENUM [18]. The analyzed algorithm belongs to the neighbor computation class (see Figure 4). In order to model real program behavior, the

execution of each task in our example is assumed as a Erlang–8 distribution. The domain to be calculated is a matrix consisting of 10 columns. If we assume that a column is the smallest unit of work that can be distributed among the processors, by default $P = 10$. The parallelization for 2 and 3 processors is depicted as an example in Figure 13 for two iterations. Each node in this figure represents a row iteration with rows numbered from left to right. Using two processors, there are inter–processor dependencies only between row 5 and 6. Using more processors, more inter–processor dependencies arise. The general model and the scaled model can be found in Figure 4.

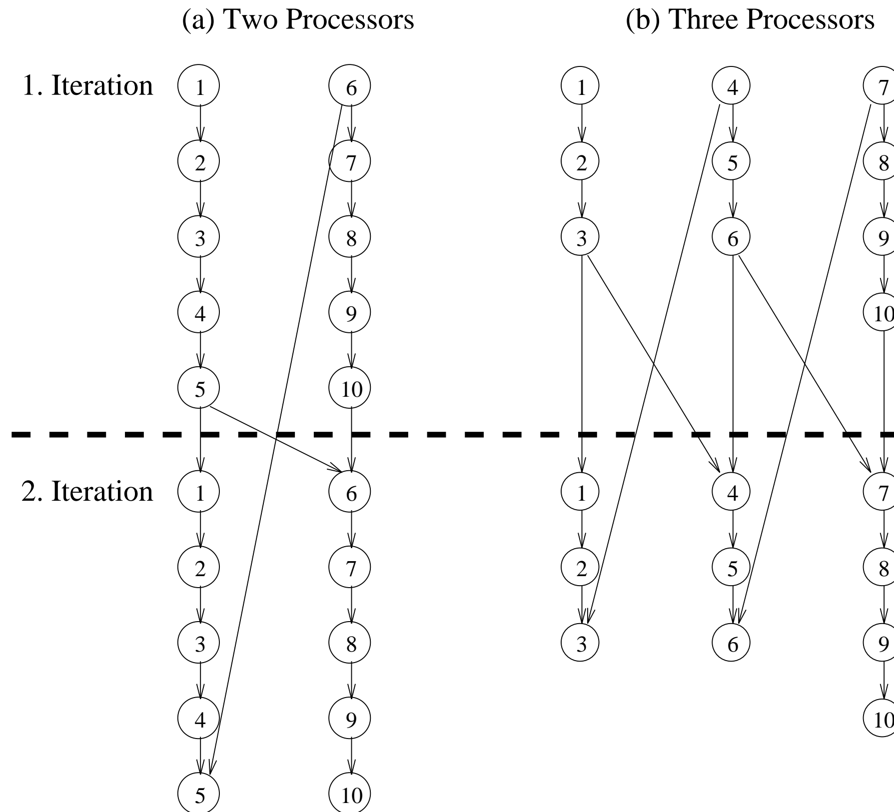


Figure 13: Stochastic Graph Models for Two Iterations

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**[2]
   Due to state space explosion, Markov analysis is not possible for more than 4 processors, even when reducing the number of iterations to 2 (Figure 13). For this case the state space is about 160,000 states, and model solution takes about 9 hours on a HP 715 workstation. Although PEPP solves graph models using an approximate state space analysis, model evaluation using state space analysis is not possible for a higher degree of parallelism (Table 1). It should be emphasized that the approximate state space analysis implemented in PEPP can deal with any runtime distribution (e.g. numerical distributions obtained from

[2]The presented numbers are modeled execution times and do not have a specific time unit

monitoring existing programs). Since evaluating large models (especially models with a high degree of parallelism causing a lot of task dependences) is hard, this method is not applicable in practice.

P	2 iterations	4 iterations	8 iterations
1	160	320	640
2	85	167.7	331.6
3	64.6	too high computations costs	
4	52.5		

Table 1: Results of State Space Analysis

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 [17]. Results obtained with bounding methods implemented in PEPP are shown in Table 2 and Figure 14. In [7] we have shown for various graph structures that the bounding methods implemented in PEPP are very accurate. In PEPP, three different bounding methods are implemented in order to select the best bound. Depending on the structure of the graph model, one or the other bounding method will yield the best result. It can be seen in the figure that the difference between upper and lower bound is very small. In some cases the computed bounds are equal to the state space analysis results.

3. **Approximate modeling**
   Approximate models (like the model shown in Figure 4) can easily be evaluated if they have a series–parallel structure. In this case a runtime distribution can be calculated using the operators *series reduction* (i.e., convolution of two densities functions) and *parallel reduction* (i.e., product of two distribution functions) to reduce the graph to one single node. Here, generation of a state space is avoided.

P	2 iterations	4 iterations	8 iterations
1	160	320	640
2	85 - 86.8	167.1 - 172.7	330 - 341.8
3	64.5 - 65.8	128.1 - 130.4	256 - 258.8
4	51.9 - 53.9	101.5 - 105.1	200 - 209
5	38.8 - 38.9	73.5 - 77.8	141 - 154
6	37.7 - 38.8	72.2 - 77.7	135 - 153
7	36.8 - 37.2	70.8 - 75.3	138 - 149
8	35.2 - 36.7	68.5 - 70.3	134 - 142
9	32.2 - 32.3	64 - 64.2	128 - 130
10	23.7 - 24.7	44.7 - 48.1	85.8 - 96.1

Table 2: Results of Bounding Methods

P	2 iterations	4 iterations	8 iterations
1	160	320	640
2	87.1	174.2	348.4
3	65.8	131.6	263.2
4	53.8	107.6	215.2
5	41.7	83.4	166.8
6	40.5	81	162
7	38.9	77.8	155.6
8	36.7	73.4	146.8
9	33	66	132
10	25.7	51.4	102.8

Table 3: Results of Approximate Modeling

Results obtained by evaluating our scaled models are upper bounds of the mean runtime. Kleinöder has proved [12] that an upper bound is obtained by inserting arcs. In this case,

the added barrier synchronization arcs are causing higher execution times. The deletion of arcs leads to a lower bound, because execution constraints, with respect to the original model, may be violated.

The results obtained from evaluating our scaled models differ only slightly from the results calculated with the sophisticated bounding methods implemented in PEPP (Table 3). Comparing the results of both methods (Table 2 and Table 3), we see that results obtained with approximate modeling estimate the runtime very well.

The speedup values presented in Figure 14 (right) verify the necessity of a systematic scalability analysis. To obtain these results with measurements, 30 measurements must be taken. For configurations of 2, 5, and 10 processors good speedup values are reached, since in these cases a good load balancing of the 10 tasks can be obtained.

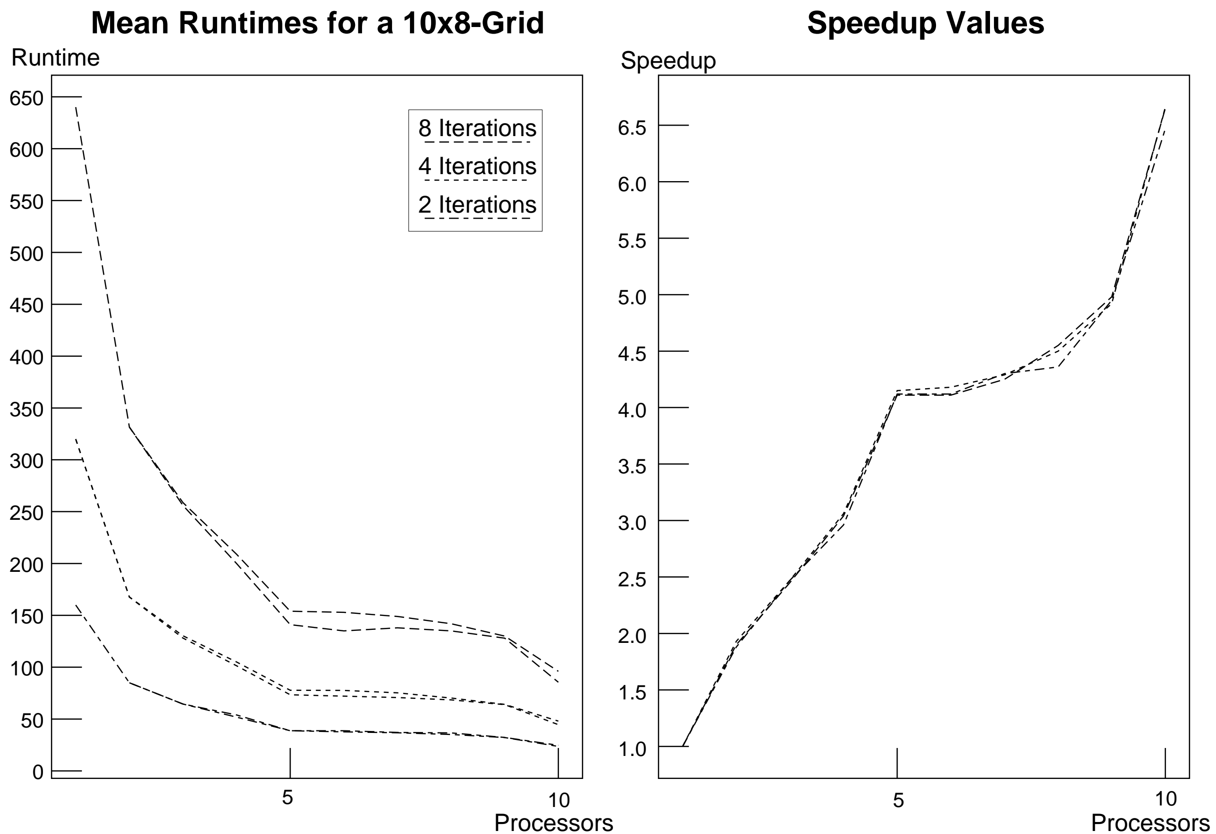


Figure 14: Scalability Analysis Results

Note, in this example, communication delays are not considered. Using a model generator, this can easily be done by substituting each inter–processor arc by an arc, a node representing communication costs, and another arc. Communication delays depend on the underlying parallel system and are specified in the machine models. The communication volume for each data exchange is specified in PDL.

Using PDL, the problem size, the runtime distributions of all tasks and communication nodes, and the underlying synchronization mechanism can easily be varied. This is a prerequisite for a flexible and systematic scalability analysis. This first results of model–based scalability analysis

show that our approach is well–suited to compare different scales of parallel systems and input data.

6 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.

By analyzing various computation classes, we have shown how scaled models can be created from a generic computation description in PDL and analyzed by an existing 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 on how problem size scaling is handled or how the machine model interacts with the analysis. We are currently exploring these issue more thoroughly through the analysis of additional testcases. Another important problem is that we can handle only static computations, because the presented computation classes depend on knowing something about the structure of computation. 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] M. Ajmone Marsan, G. Balbo, and G. Conte. A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems. *ACM Transactions on Computer Systems*, 2(2):93–122, May 1984.

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

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

[4] P. Dauphin, F. Hartleb, M. Kienow, V. Mertsiotakis, and A. Quick. PEPP: Performance Evaluation of Parallel Programs — User's Guide – Version 3.3. Technical Report 17/93, Universität Erlangen–Nürnberg, IMMD VII, September 1993.

[5] J. Davies. Parallel Loop Constructs for Multiprocessors. Master's thesis, UIUC-CS, 1981.

[6] Z. Fang, P. Yew, P. Tang, and C. Zhu. Dynamic Processor Self-Scheduling for General Parallel Nested Loops. In *ICPP-87*, pages 1–10, 1987.

[7] 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.

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

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

[10] U. Herzog and W. Hofmann. Synchronization Problems in Hierarchically Organized Multiprozessor Computer Systems. In M. Arato, A. Butrimenko, and E. Gelenbe, editors, *Performance of Computer Systems – Proceedings of the , 4th International Symposium on Modelling and Performance Evaluation of Computer Systems*, Vienna, Austria, Februar, 6–8 1979.

[11] 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.

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

[13] A.D. Malony, V. Mertsiotakis, A. Quick. Stochastic Modeling of Scaled Parallel Programs. Technical Report, Universität Erlangen–Nürnberg, IMMD VII, 1994.

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

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

[16] 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.

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

[18] U. Trottenberg. (ed). Special Issue on the 2nd International SUPRENUM Colloquium. *Parallel Computing*, 7, 1988.

[19] H. Wabnig, G. Kotsis, and G. Haring. Performance Prediction of Parallel Programs. In B. Walke and O. Spaniol, editors, *Proceedings der 7. GI–ITG Fachtagung "Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen", Aachen, 21.–23. Spetember 1993*, pages 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 Transactions on Software Engineering*, 17(10):1005–1012, October 1991.