EVALUATING THE PERFORMANCE OF PIPELINE-STRUCTURED PARALLEL PROGRAMS WITH SKELETONS AND PROCESS ALGEBRA*

ANNE BENOIT†, MURRAY COLE , STEPHEN GILMORE , AND JANE HILLSTON

Abstract. We show in this paper how to evaluate the performance of pipeline-structured parallel programs with skeletons and process algebra. Since many applications follow some commonly used algorithmic skeletons, we identify such skeletons and model them with process algebra in order to get relevant information about the performance of the application, and to be able to take good scheduling decisions. This concept is illustrated through the case study of the pipeline skeleton, and a tool which generates automatically a set of models and solves them is presented. Some numerical results are provided, proving the efficacy of this approach.

Key words. Algorithmic skeletons, pipeline, high-level parallel programs, performance evaluation, process algebra, PEPA Workbench.

1. Introduction. One of the most promising technical innovations in present-day computing is the invention of grid technologies which harness the computational power of widely distributed collections of computers [8]. Designing an application for the Grid raises difficult issues of resource allocation and scheduling (roughly speaking, how to decide which computer does what, and when, and how they interact). These issues are made all the more complex by the inherent unpredictability of resource availability and performance. For example, a supercomputer may be required for a more important task, or the Internet connections required by the application may be particularly busy.

In this context of grid programming, a skeleton-based approach [5, 16, 7] recognizes that many real applications draw from a range of well-known solution paradigms and seeks to make it easy for an application developer to tailor such a paradigm to a specific problem. Powerful structuring concepts are presented to the application programmer as a library of pre-defined 'skeletons'. As with other high-level programming models the emphasis is on providing generic polymorphic routines which structure programs in clearly-delineated ways. Skeletal parallel programming supports reasoning about parallel programs in order to remove programming errors. It enhances modularity and configurability in order to aid modification, porting and maintenance activities. In the present work we focus on the Edinburgh Skeleton Library (eSkel) [6]. eSkel is an MPI-based library which has been designed for SMP and cluster computing and is now being considered for grid applications using grid-enabled versions of MPI such as MPICH-G2 [14].

The use of a particular skeleton carries with it considerable information about implied scheduling dependencies. By modelling these with stochastic process algebras such as Performance Evaluation Process Algebra [13], and thereby being able to include aspects of uncertainty which are inherent to grid computing, we believe that we will be able to underpin systems which can make better scheduling decisions than less sophisticated approaches. Most significantly, since this modelling process can be automated, and since grid technology provides facilities for dynamic moni-

*This work is part of the ENHANCE project, funded by the United Kingdom Engineering and Physical Sciences Research council grant number GR/S21717/01.

†School of Informatics, The University of Edinburgh, James Clerk Maxwell Building, The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, UK. enhancers@inf.ed.ac.uk, http://groups.inf.ed.ac.uk/enhance/

toring of resource performance, our approach will support *adaptive* rescheduling of applications.

Stochastic process algebras were introduced in the early 1990s as a compositional formalism for performance modelling. Since then they have been successfully applied to the analysis of a wide range of systems. In general analysis is based on the generation of an underlying continuous time Markov chain (CTMC) and derivation of its steady state probability distribution. This vector records the likelihood of each potential state of the system, and can in turn be used to derive performance measures such as throughput, utilisation and response time. Several stochastic process algebras have appeared in the literature; we use Hillston's Performance Evaluation Process Algebra (PEPA) [13].

Some related projects obtain performance information from the Grid using benchmarking and monitoring techniques [4, 17]. In the ICENI project [9], performance models are used to improve the scheduling decisions, but these are just graphs which approximate data obtained experimentally. Moreover, there is no upper-level layer based on skeletons in any of these approaches.

Other recent work considers the use of skeleton programs within grid nodes to improve the quality of cost information [1]. Each server provides a simple function capturing the cost of its implementation of each skeleton. In an application, each skeleton therefore runs only on one server, and the goal of scheduling is to select the most appropriate servers within the wider context of the application and supporting grid. In contrast, our approach considers single skeletons which span the Grid. Moreover, we use modelling techniques to estimate performance.

Our main contribution is based on the idea of using performance models to enhance the performance of grid applications. We propose to model skeletons in a generic way to obtain significant performance results which may be used to reschedule the application dynamically. To the best of our knowledge, this kind of work has not been done before. We show in this paper how we can obtain significant results on a first case study based on the pipeline skeleton. An earlier version of this paper is published in the proceedings of the workshop on Practical Aspects of High-level Parallel Programming (PAPP04), part of the International Conference on Computational Science (June 7-9, 2004, Kraków, Poland) [2]. In this extended version a presentation of PEPA is included; the model resolution and the tool AMoGeT are described more precisely; and more experimental results are exposed.

In the next section, we present the pipeline and a model of the skeleton. Then we explain how to solve the model with the PEPA Workbench in order to get relevant information (Section 3). In Section 4 we present a tool which automatically determines the best mapping to use for the application, by first generating a set of models, then solving them and comparing the results. Some numerical results on the pipeline application are provided in Section 5, and the feasibility of this approach is discussed in Section 6. Finally we give some conclusions.

2. The pipeline skeleton. Many parallel algorithms can be characterized and classified by their adherence to one or more of a number of generic algorithmic skeletons [16, 5, 7]. We focus in this paper on the concept of pipeline parallelism, which is of well-proven usefulness in several applications. We recall briefly the principle of the pipeline skeleton. Then we introduce the process algebra PEPA [13] and we explain how we can model the pipeline with PEPA. Finally, we show in Section 2.4 the state transition diagram of a three stage pipeline.

2.1. The principle of pipeline. In the simplest form of pipeline parallelism [6], a sequence of N_s *stages* process a sequence of *inputs* to produce a sequence of *outputs* (Fig. 2.1).

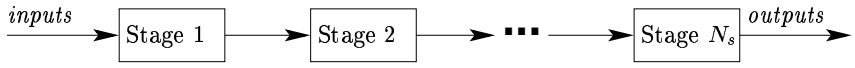

FIG. 2.1. *The pipeline application*

Each input passes through each stage in the same order, and the different inputs are processed one after another (a stage cannot process several inputs at the same time). Note that the internal activity of a stage may be parallel, but this is transparent to our model. In the remainder of the paper we use the term "processor" to denote the hardware responsible for executing such activity, irrespective of its internal design (sequential or parallel).

We consider this application class in the context of computational grids, and so we want to map it to our computing resources, which consist of a set of potentially heterogeneous processors interconnected by a heterogeneous network.

It is well known that a computing pipeline performs most effectively when the workload is well balanced across stages and there are a large enough number of inputs to amortize the costs of filling and draining. Our work directly addresses the first of these issues, by facilitating exploration of the stage-to-processor mapping space. The second issue remains the responsibility of the programmer: our approach assumes that running the application will take long enough for the system to reach an equilibrium behaviour. The models help us to study this steady state behaviour.

Considering the pipeline application in the eSkel library [6], we focus here on a pipeline variant which requires that each stage produces exactly one output for each input.

We now go on to present the PEPA language which we will use to model the pipeline application. The presentation below is necessarily brief and rather informal. For full details the reader is referred to [13]. The operational semantics can also be found in Appendix A.

2.2. Introduction to PEPA. The PEPA language provides a small set of combinators. These allow language terms to be constructed defining the behaviour of components, via the activities they undertake and the interactions between them. Timing information is associated with each activity. Thus, when enabled, an activity $a = (\alpha, r)$ will delay for a period sampled from the negative exponential distribution which has parameter r. If several activities are enabled concurrently, either in competition or independently, we assume that a *race condition* exists between them. The component combinators, together with their names and interpretations, are presented informally below.

Prefix: The basic mechanism for describing the behaviour of a system is to give a component a designated first action using the prefix combinator, denoted by a full stop. For example, the component $(\alpha, r).S$ carries out activity (α, r), which has action type α and an exponentially distributed duration with parameter r, and it subsequently behaves as S.

Choice: The choice combinator captures the possibility of competition between different possible activities. The component $P+Q$ represents a system which may behave either as P or as Q. The activities of both P and Q are enabled. The first activity to

complete distinguishes one of them: the other is discarded. The system will behave as the derivative resulting from the evolution of the chosen component.

Constant: It is convenient to be able to assign names to patterns of behaviour associated with components. Constants are components whose meaning is given by a defining equation. For example, $P \stackrel{def}{=} (\alpha, r).P$ defines a component which performs activity α at rate r, forever.

Hiding: The possibility to abstract away some aspects of a component's behaviour is provided by the hiding operator, denoted P/L. Here, the set L of visible action types identifies those activities which are to be considered internal or private to the component and which will appear as the unknown type τ.

Cooperation: In PEPA direct interaction, or *cooperation*, between components is the basis of compositionality. The set which is used as the subscript to the cooperation symbol, the *cooperation set* L, determines those activities on which the *co-operands* are forced to synchronise. For action types not in L, the components proceed independently and concurrently with their enabled activities. However, an activity whose action type is in the cooperation set cannot proceed until both components enable an activity of that type. The two components then proceed together to complete the *shared activity*. The rate of the shared activity may be altered to reflect the work carried out by both components to complete the activity (for details see [13]). We write $P \parallel Q$ as an abbreviation for $P \underset{L}{\bowtie} Q$ when L is empty.

In some cases, when an activity is known to be carried out in cooperation with another component, a component may be *passive* with respect to that activity. This means that the rate of the activity is left unspecified (denoted $\top$) and is determined upon cooperation, by the rate of the activity in the other component. All passive actions must be synchronised in the final model.

The dynamic behaviour of a PEPA model is represented by the evolution of its components, either individually or in cooperation. The form of this evolution is governed by a set of formal rules which give an operational semantics of PEPA terms (see [13]). Thus, as in classical process algebra, the semantics of each term in PEPA is given via a labelled *multi-transition* system (the multiplicities of arcs are significant). In the transition system a state corresponds to each syntactic term of the language, or *derivative*, and an arc represents the activity which causes one derivative to evolve into another. The complete set of reachable states is termed the *derivative set* of a model and these form the nodes of the *derivation graph* which is formed by applying the semantic rules exhaustively.

The derivation graph is the basis of the underlying Continuous Time Markov Chain (CTMC) which is used to derive performance measures from a PEPA model. The graph is systematically reduced to a form where it can be treated as the state transition diagram of the underlying CTMC. Each derivative is then a state in the CTMC. The *transition rate* between two derivatives P and Q in the derivation graph is the rate at which the system changes from behaving as component P to behaving as Q. It is the sum of the activity rates labelling arcs connecting node P to node Q.

2.3. Pipeline model. To model a pipeline application, we decompose the problem into the stages, the processors and the network. The model is expressed in PEPA (c.f. Section 2.2).

The stages

The first part of the model is the *application model*, which is specified independently

of the resources on which the application will be computed. We define one PEPA component per stage. For $i = 1..N_s$, the component $Stage_i$ works sequentially. At first, it gets data (activity $move_i$), then processes it (activity $process_i$), and finally moves the data to the next stage (activity $move_{i+1}$).

$$Stage_i \stackrel{def}{=} (move_i, \top).(process_i, \top).(move_{i+1}, \top).Stage_i$$

All the rates are unspecified, denoted by the distinguished symbol $\top$, since the processing and move times depend on the resources where the application is running. These rates will be defined later, in another part of the model.

The pipeline application is then defined as a cooperation of the different stages over the $move_i$ activities, for $i = 2..N_s$.

The activities $move_1$ and $move_{N_s+1}$ represent, respectively, the arrival of an input in the application and the transfer of the final output out of the pipeline. They do not represent any data transfer between stages, so they are not synchronizing the pipeline application. Finally, we have:

$$Pipeline \stackrel{def}{=} Stage_1 \underset{\{move_2\}}{\bowtie} Stage_2 \underset{\{move_3\}}{\bowtie} \ldots \underset{\{move_{N_s}\}}{\bowtie} Stage_{N_s}$$

The processors

We consider that the application must be mapped on a set of N_p processors. Each stage is processed by a given (unique) processor, but a processor may process several stages (in the case where $N_p < N_s$). In order to keep the model simple, we decide to put information about the processor (such as the load of the processor or the number of stages being processed) directly in the rate μ_i of the activities $process_i$, $i = 1..N_s$ (these activities have been defined for the components $Stage_i$).

Each processor is then represented by a PEPA component which has a cyclic behaviour, consisting of processing sequentially inputs for a stage. Some examples follow.

- In the case when $N_p = N_s$, we map one stage per processor:

$$Processor_i \stackrel{def}{=} (process_i, \mu_i).Processor_i$$

- If several stages are processed by a same processor, we use a choice composition. In the following example ($N_p = 2$ and $N_s = 3$), the first processor processes the two first stages, and the second processor processes the third stage.

$$\begin{aligned} Processor_1 &\stackrel{def}{=} (process_1, \mu_1).Processor_1 + (process_2, \mu_2).Processor_1 \\ Processor_2 &\stackrel{def}{=} (process_3, \mu_3).Processor_2 \end{aligned}$$

Since all processors are independent, the set of processors is defined as a parallel composition of the processor components:

$$Processors \stackrel{def}{=} Processor_1 || Processor_2 || \ldots || Processor_{N_p}$$

The network

The last part of the model is the network. We do not need to directly model the architecture and the topology of the network for what we aim to do, but we want to get some information about the efficiency of the link connection between pairs of

processors. This information is given by affecting the rates λ_i of the $move_i$ activities ($i = 1..N_s + 1$).

- λ_1 represents the connection between the user (providing inputs to the pipeline) and the processor hosting the first stage.
- For $i = 2..N_s$, λ_i represents the connection between the processor hosting stage $i-1$ and the processor hosting stage i.
- λ_{N_s+1} represents the connection between the processor hosting the last stage and the user (the site where we want the output to be delivered).

Note that λ_i will encode information both about the load on the links and the size of the data processed by $process_{i-1}$. When the data is "transferred" on the same computer, the rate is really high, meaning that the connection is fast (compared to a transfer between different sites).

The network is then modelled by the following component:

$$Network \stackrel{def}{=} (move_1, \lambda_1).Network + \ldots + (move_{N_s+1}, \lambda_{N_s+1}).Network$$

The pipeline model

Once we have defined the different components of our model, we just have to map the stages onto the processors and the network by using the cooperation combinator. For this, we define the following sets of action types:

- $L_p = \{process_i\}_{i=1..N_s}$ to synchronize the $Pipeline$ and the $Processors$
- $L_m = \{move_i\}_{i=1..N_s+1}$ to synchronize the $Pipeline$ and the $Network$

$$Mapping \stackrel{def}{=} Network \underset{L_m}{\bowtie} Pipeline \underset{L_p}{\bowtie} Processors$$

PEPA input file

An example of an input file for the PEPA Workbench can be found in Appendix B.

2.4. State transition diagram for the pipeline model. Figure 2.2 represents the state transition diagram of a three stage, three process pipeline. This picture shows all of the possible interleavings of the components of the model with arcs of various kinds showing the different types of transitions from state to state.

In Table 2.1 we show the correspondence between the state numbers in Figure 2.2 and the PEPA terms. Since the PEPA terms are long we have omitted the cooperation sets, showing only the local state of each component. Moreover to keep the table compact we have named the derivatives of the $Stage$ components as follows:

$$Stage_{i0} \stackrel{def}{=} (move_i, \top).Stage_{i1}$$
$$Stage_{i1} \stackrel{def}{=} (process_i, \top).Stage_{i2}$$
$$Stage_{i2} \stackrel{def}{=} (move_{i+1}, \top).Stage_{i0}$$

3. Solving the models. One reason to work with a formal modelling language such as PEPA is that models are unambiguous and can serve to support reliable communication between those who design systems, those who develop them and those who maintain them. Another reason to work with a formal modelling language is that formal models can be automatically processed by tools in order to derive information from them which otherwise would have to be produced by manual calculation or reasoning.

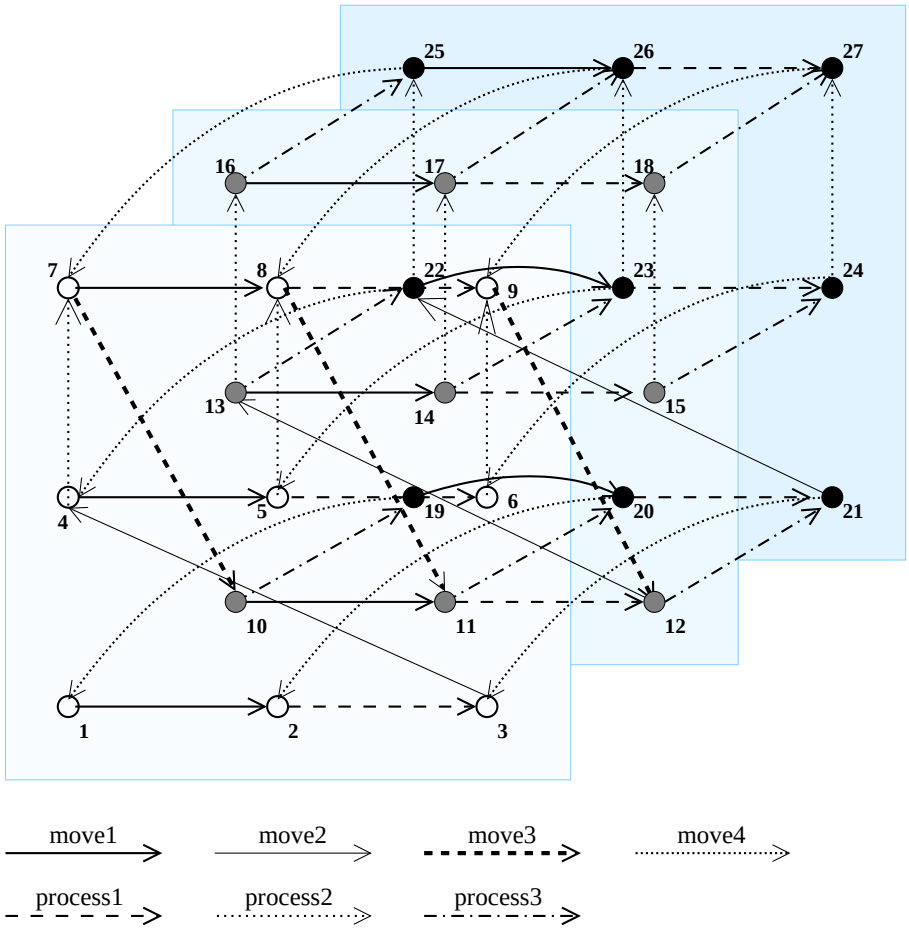

FIG. 2.2. *State transition diagram of a three stage, three process pipeline with states numbered according to Table 2.1*

The tool which we have used for processing our PEPA models and computing the steady-state probability distribution of our system is the PEPA Workbench. A full description of the functioning of this software can be found in [11]; the reference manual for the latest release is [12]. We include a brief description of the functioning of the Workbench in Appendix C.1 in order to make the present paper self-contained.

Notice however that the steady-state probability distribution of the system is rarely the desired result of the performance analysis process and so to progress we must identify a significant *performance result*. The performance result that is pertinent for the pipeline application is the throughput of the $process_i$ activities ($i = 1..N_s$). Since data passes sequentially through each stage, the throughput is identical for all i, and we need to compute only the throughput of $process_1$ to obtain significant results. This is done by adding the steady-state probabilities of each state in which $process_1$ can happen, and multiplying this by μ_1.

We have made some changes to the Java edition of the PEPA Workbench in order to allow the user to specify performance results which will then be automatically computed. This new functionality is then used to compute numerical results from the

state no.	*PEPA state*
1	$(Network, (Stage_{10}, Stage_{20}, Stage_{30}), (Processor1, Processor2, Processor3))$
2	$(Network, (Stage_{11}, Stage_{20}, Stage_{30}), (Processor1, Processor2, Processor3))$
3	$(Network, (Stage_{12}, Stage_{20}, Stage_{30}), (Processor1, Processor2, Processor3))$
4	$(Network, (Stage_{10}, Stage_{21}, Stage_{30}), (Processor1, Processor2, Processor3))$
5	$(Network, (Stage_{11}, Stage_{21}, Stage_{30}), (Processor1, Processor2, Processor3))$
6	$(Network, (Stage_{12}, Stage_{21}, Stage_{30}), (Processor1, Processor2, Processor3))$
7	$(Network, (Stage_{10}, Stage_{22}, Stage_{30}), (Processor1, Processor2, Processor3))$
8	$(Network, (Stage_{11}, Stage_{22}, Stage_{30}), (Processor1, Processor2, Processor3))$
9	$(Network, (Stage_{12}, Stage_{22}, Stage_{30}), (Processor1, Processor2, Processor3))$
10	$(Network, (Stage_{10}, Stage_{20}, Stage_{31}), (Processor1, Processor2, Processor3))$
11	$(Network, (Stage_{11}, Stage_{20}, Stage_{31}), (Processor1, Processor2, Processor3))$
12	$(Network, (Stage_{12}, Stage_{20}, Stage_{31}), (Processor1, Processor2, Processor3))$
13	$(Network, (Stage_{10}, Stage_{21}, Stage_{31}), (Processor1, Processor2, Processor3))$
14	$(Network, (Stage_{11}, Stage_{21}, Stage_{31}), (Processor1, Processor2, Processor3))$
15	$(Network, (Stage_{12}, Stage_{21}, Stage_{31}), (Processor1, Processor2, Processor3))$
16	$(Network, (Stage_{10}, Stage_{22}, Stage_{31}), (Processor1, Processor2, Processor3))$
17	$(Network, (Stage_{11}, Stage_{22}, Stage_{31}), (Processor1, Processor2, Processor3))$
18	$(Network, (Stage_{12}, Stage_{22}, Stage_{31}), (Processor1, Processor2, Processor3))$
19	$(Network, (Stage_{10}, Stage_{20}, Stage_{32}), (Processor1, Processor2, Processor3))$
20	$(Network, (Stage_{11}, Stage_{20}, Stage_{32}), (Processor1, Processor2, Processor3))$
21	$(Network, (Stage_{12}, Stage_{20}, Stage_{32}), (Processor1, Processor2, Processor3))$
22	$(Network, (Stage_{10}, Stage_{21}, Stage_{32}), (Processor1, Processor2, Processor3))$
23	$(Network, (Stage_{11}, Stage_{21}, Stage_{32}), (Processor1, Processor2, Processor3))$
24	$(Network, (Stage_{12}, Stage_{21}, Stage_{32}), (Processor1, Processor2, Processor3))$
25	$(Network, (Stage_{10}, Stage_{22}, Stage_{32}), (Processor1, Processor2, Processor3))$
26	$(Network, (Stage_{11}, Stage_{22}, Stage_{32}), (Processor1, Processor2, Processor3))$
27	$(Network, (Stage_{12}, Stage_{22}, Stage_{32}), (Processor1, Processor2, Processor3))$

TABLE 2.1

Correspondence between state numbers in Figure 2.2 and PEPA terms (cooperation sets are omitted but remain constant)

pipeline models. Some more technical details are provided in Appendix C.2.

4. AMoGeT: The Automatic Model Generation Tool. We investigate in this paper how to enhance the performance of grid applications with the use of algorithmic skeletons and process algebras. To do this, we have created a tool which automatically generates performance models for the pipeline case study, and then solves the models. These results could be used to reschedule the application.

We give at first an overview of the tool. Then we describe the information which is provided to the tool via a *description file*. Finally, we explain the functioning of the tool.

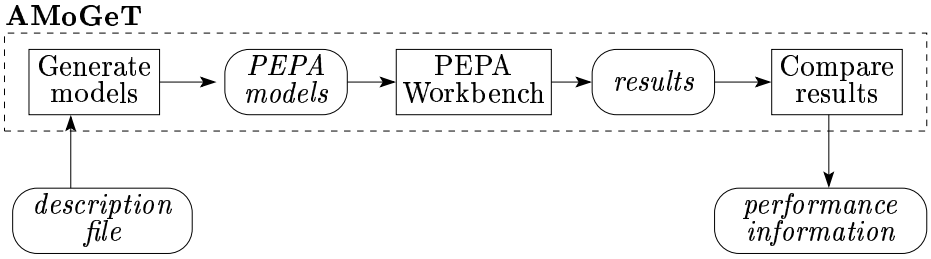

FIG. 4.1. *The principle of AMoGeT*

4.1. AMoGeT description. Fig. 4.1 illustrates the principle of the tool. In its current form, the tool is a generic, reusable software component. Its ultimate role will be as an integrated component of a run-time scheduler and re-scheduler, adapting the mapping from application to resources in response to changes in resource availability and performance.

Information is provided to the tool via a *description file* (c.f. Section 4.2). This information can be gathered from the Grid resources and from the application definition. In the following experiments, it is provided by the user, but we can also get it automatically from grid services, for example from the Network Weather Service [17].

The tool allows everything to be done in a single step through a simple Perl script (c.f. Section 4.3): it generates the models, solves them with the PEPA Workbench, and then compares the results. This allows us to have feedback on the application when the performance of the available resources is modified.

4.2. Description file for AMoGeT. The aim of this file is to provide information about the available grid resources and the modelled application, in our case the pipeline.

This description file is named `mymodel.des`, where *mymodel* is the name of the application.

- The first information provided is the type of the model. Since we study here the pipeline skeleton, the first line is
  `type` = `pipeline`;
- We then have the information about the Grid resources and Network links, as a list of parameters. The number of processors N must at first be specified:
  `nbproc` $=N$;
  And then, for $i = 1..N$ and $j = 1..N$, we specify the available computing power of the processor i (`cp`i), and the performance of the network link between processors i and j (`nl`i`-`j):
  `cp1=10; cp2=5;`
  `nl1-1=10000; nl1-2=8;`
  `cp`i captures the fact that a processor's full power may not be available to our application (e.g. because of time-sharing with other activities).
- Concerning the application, we have some information about the stages of the pipeline. N_s is the number of stages.
  `nbstage=`N_s`;`
  The amount of work `w`i required to compute one output for stage i must be specified for $i = 1..N_s$:
  `w1=2; w2=4; ...`
  Finally, we need to specify the size of the data transferred to and from each stage. For $i = 1..N_s + 1$, `ds`i is the size of the data transferred to stage i, with the boundary case `ds`$N_s + 1$ which represents the size of the output data.
  `ds1=100; ds2=5; ...`
- Next we define a set of candidate mappings of stages to processors. Each mapping specifies where the initial data is located, where the output data must be left and (as a tuple) the processor where each stage is processed. For example, the tuple $(1, 1, 2)$ means that the two first stages are on processor 1, with the third stage on processor 2. A mapping is then of the form $[input, tuple, output]$. The mapping definition is a set of mappings, it can be as follows:

```
mappings=[1,(1,2,3),3],[1,(1,1,2),2],[1,(1,1,1),1];
```

- The last thing is the performance result we want to compute. For the pipeline application, we can ask for the *throughput* with the line:
  ```
  throughput;
  ```

4.3. The AMoGeT Perl script. The tool allows everything to be done in a single step through a simple Perl script. The model generation is done by calling an auxiliary function. Models are then solved with the PEPA Workbench as seen in Section 3. Finally, the results are compared. This allows us to have feedback on the application when the performance of the available resources is modified.

One model is generated from each mapping of the description file. Each model is as described in Section 2.3. The difficult point consists of generating the rates from the information gathered before. The model generation itself is then straightforward.

To compute the rates of the $process_i$ activities for a given model ($i = 1..N_s$), we need to know how many stages are hosted on each processor, and we assume that the work sharing between the stages is equitable. The rate associated with the $process_i$ activity is then:

$$\mu_i = \mathtt{w}i \times \frac{\mathtt{cp}j}{\mathtt{nbst}j}$$

where j is the number of the processor hosting the stage i, and $\mathtt{nbst}j$ is the number of stages being processed on processor j. In effect, the available computing power $\mathtt{cp}j$ is further diluted by our own internal timesharing factor $\mathtt{nbst}j$, before being applied to the workload associated with the stage, $\mathtt{w}i$.

The rates of communication between stages depend on the mapping too, since the rate of a $move_i$ activity depends on the connection link between the processor j_1 hosting stage $i-1$ and the processor j_2 hosting stage i, which is given by $\mathtt{nl}j_1\text{-}j_2$. Since the mapping specifies where the input and output data are, we can also find the connection link for the data arriving into the pipeline and the data exiting the application. These rates depend also on the size of the data transferred from one stage of the pipeline to the next, given by $\mathtt{ds}i$. The boundary cases are applied to compute the rates of the $move_1$ and $move_{N_s+1}$ activities. The rate associated with the $move_i$ activity is therefore:

$$\lambda_i = \frac{\mathtt{nl}j_1\mathtt{-j}_2}{\mathtt{ds}i}$$

Once these rates are derived, generating the model is straightforward. We add into the file the description of the throughput of the $process_1$ activity as a required result to allow an automatic computation of this result. The models can then be solved with the PEPA Workbench, and the throughput of the pipeline is automatically computed (Section 3). During the resolution, all the results are saved in a single file, and the last step of results comparison finds out which mapping produces the best throughput. This mapping is the one we should use to run the application.

5. Numerical results. We present in this section some numerical results. We explain through them how the information obtained with AMoGeT can be relevant for optimizing the application.

In the present paper we do not apply this method to a given "real-world" example. We use an abstract pipeline for which we arbitrarily fix the time required to complete each stage. This is sufficient to show that AMoGeT can help to optimize an application.

5.1. Experiment 1: Pipeline with 3 stages - fixed data size. We give here a few numerical results on an example with 3 pipeline stages (and up to 3 processors). The models that we need to solve are really small (in this case, the model has 27 states and 51 transitions, c.f Figure 2.2).

We suppose in this experiment that `nli-i=10000` for $i = 1..3$, and that there is no need to transfer the input or the output data. Moreover, we suppose that the network is symmetrical (`nli-j=nlj-i` for all $i, j = 1..3$). Concerning the pipeline parameters, the amount of work `wi` required to compute each stage is 1, as well as the size of the data `dsi` which is transferred from one stage to another. The relevant parameters are therefore `nl1-2`, `nl2-3`, `nl1-3`, and `cpi` for $i = 1..3$. We compare different mappings, and just specify the tuple indicating which stage is on which processor. We compare the mappings (1,1,1), (1,1,2), (1,1,3), (1,2,1), (1,2,2), (1,2,3), (1,3,1), (1,3,2) and (1,3,3) (the first stage is always on processor 1). The results are displayed in Table 5.1, and we only put the best of the mappings which were investigated in the relevant line of the table.

Set of results	Parameters						Mapping & Throughput
	`nl1-2`	`nl2-3`	`nl1-3`	`cp1`	`cp2`	`cp3`	
1	10000	10000	10000	10	10	10	(1,2,3): 5.63467
	10000	10000	10000	5	5	5	(1,2,3): 2.81892
2	10000	10000	10000	10	10	1	(1,2,1): 3.36671
	10	10	10	10	10	1	(1,1,2): 2.59914
	1	1	1	10	10	1	(1,1,1): 1.87963
3	10	1	1	10	10	10	(1,1,2): 2.59914
	10	1	1	1	1	100	(1,3,3): 0.49988

TABLE 5.1
Result table for Experiment 1

In the first set of results, all the processors are identical and the network links are really fast. In these cases, the best mapping always consists of putting one stage on each processor (the results for the mapping $(1, 3, 2)$ are identical to the best mapping). If we divide the time allocated by the processor to the application by 2, the resulting throughput is also divided by 2, since only the processing power has an impact on the throughput.

The second set of results illustrates the case when one processor is becoming really busy, in this case processor 3. We should not use it any more, but depending on the network links, the best mapping may change. If the links are not efficient, we should indeed avoid data transfer and try to put consecutive stages on the same processor. When `nl1-2` = `nl2-3` = `nl1-3` = 10, the mapping $(1, 2, 2)$ provides the same results as $(1, 1, 2)$.

Finally, the third set of results shows what happens if the network link to processor 3 is really slow. In this case again, the use of the processor should be avoided, and the best mappings are $(1, 1, 2)$ and $(1, 2, 2)$. However, if processor 3 is a really fast processor compared to the other ones (last line), we process stage 2 and stage 3 on the third processor (mapping $(1, 3, 3)$).

5.2. Experiment 2: Pipeline with 3 stages - data size changing. The third experiment keeps the 3 stage pipeline, but considers changes in the size of the data. The assumptions are the same as for Experiment 1, but more parameters have a fixed value.

In this experiment, the network connection between processors 1 and 2 is slightly less effective than the others. So, we have `nl1-2` = 100, `nl2-3` = `nl1-3` = 1000. Moreover, the computing power of each stage is `cp`i = 10. The size of the data is now fixed to 100, except from the data transiting from stage 1 to stage 2 (`ds2`), whose size is varying.

Figure 5.1 presents the throughput obtained with each mapping, as a function of the data size `ds2`.

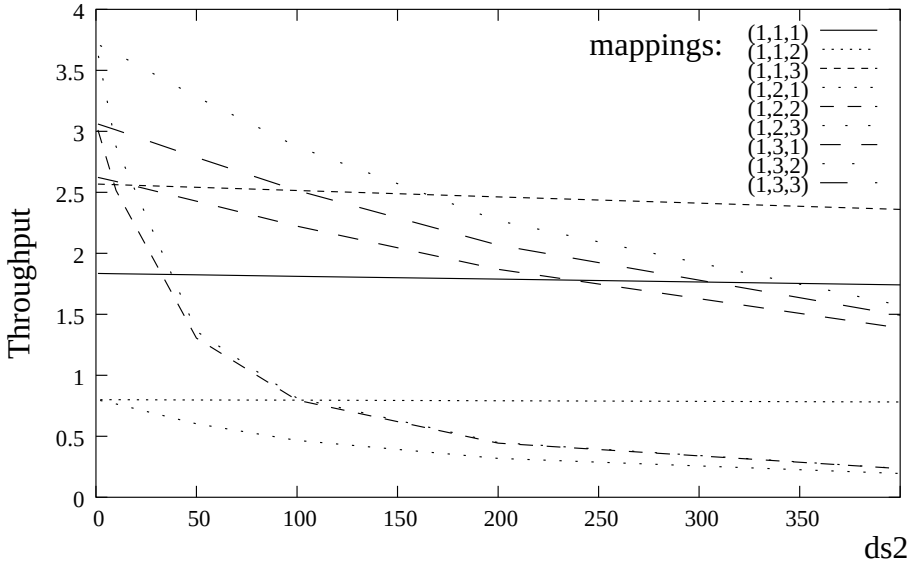

FIG. 5.1. *Experiment 2: Throughput function of* `ds2`

Notice first that some of the mappings are not influenced by the change of the data size, i.e. (1,1,1), (1,1,2) and (1,1,3). This is due to the fact that the connection between stages 1 and 2 is good because the data stays on the same processor. The influence of the size of the data transferred is much more important when the connection is less effective (mappings (1,2,2) and (1,2,3)), since the $move_2$ activity is then the bottleneck of the system.

The best mapping is (1,3,2) when `ds2` < 150, and (1,1,3) for greater values. Both of them avoid the slow connection `nl1-2`, and they use several processors so the processing power is better than for mappings like (1,1,1). When the size of the data transferred between the first two stages becomes high, the bottleneck is the connection link between them, so it is better to put them on the same processor, even if we may lose some processing power.

5.3. Experiment 3: Pipeline with 8 stages. The last experiment considers a larger pipeline, composed of 8 stages. We use up to 8 processors, and compare four different mappings, depending on the number of processors we wish to use:

- **8 processors**, the mapping is $[1, (1, 2, 3, 4, 5, 6, 7, 8), 8]$
- **4 processors**, the mapping is $[1, (1, 1, 2, 2, 3, 3, 4, 4), 4]$

- **2 processors**, the mapping is $[1,(1,1,1,1,2,2,2,2),2]$
- **1 processor**, the mapping is $[1,(1,1,1,1,1,1,1,1),1]$

The parameters are the same as for Experiment 1, with $\texttt{cp}i = 10$, $\texttt{w}i = 1$, $\texttt{ds}i\texttt{=1}$ and $\texttt{nl}i\texttt{-}i = 10000$ for all i. We vary the parameters $\texttt{nl}i\texttt{-}j$, for $i \neq j$, assuming that all these links are equal, and we compute the throughput for the different mappings. Figure 5.2 displays the results.

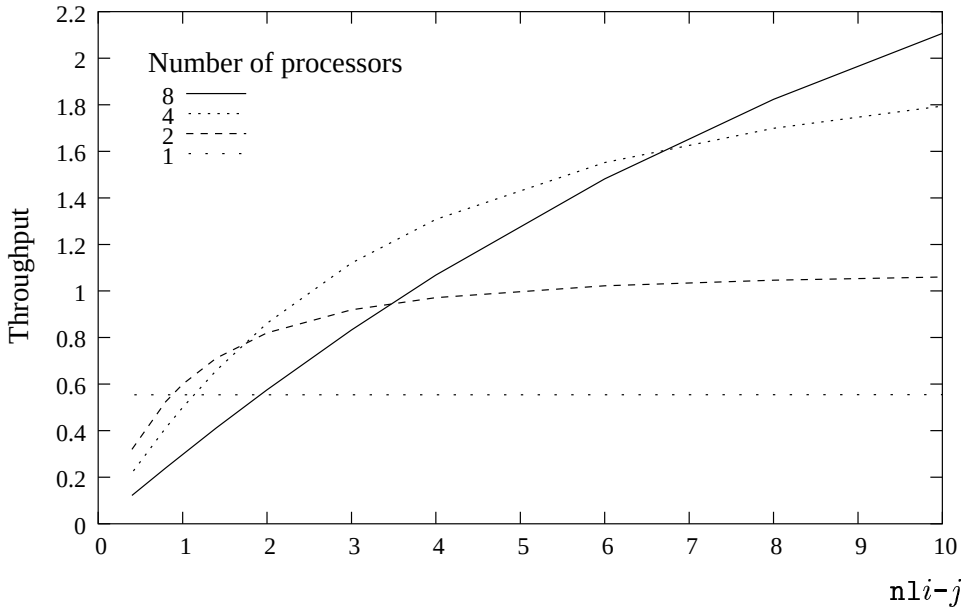

FIG. 5.2. *Experiment 3: Pipeline with* 8 *stages*

The curves obtained confirm that we should avoid data transfer when the network connections are less efficient. When $\texttt{nl}i\texttt{-}j > 7$, the network performs well enough to allow the use of the 8 processors. However, when the performance decreases, we should use only 4 processors, then two, and only one when $\texttt{nl}i\texttt{-}j < 0.8$.

When we need to transfer the output data back to the first processor (for example, the mapping $[1,(1,2,3,4,5,6,7,8),1]$ for the case with 8 processors), we obtain almost the same results, with a slightly smaller throughput due to this additional transfer.

6. Feasibility of the approach. We envisage the use of our approach within a scheduling and rescheduling platform for long-running grid applications. In this context it is anticipated that after initial analysis and scheduling, the system would be monitored and that rescheduling would be needed only relatively infrequently, for example, once an hour. Nevertheless it is important that the use of the tool does not contribute an overhead which eliminates the benefit to be obtained from its use. In this section we present evidence which suggests that this is not likely to be the case in practice. The reader should note that here we are reflecting on the performance of the analysis tools themselves rather than on the performance of the application which they monitor (as presented in the previous section).

We ran an experiment to assess the time taken to generate and solve models using AMoGeT, which will, of course, be dependent on the size of the generated model. Fig. 6.1 illustrates the number of states and transitions of the models as a function

of the parameters of the skeleton. These numbers are independent of the number of processors in the model; they depend only on the number of pipeline stages.

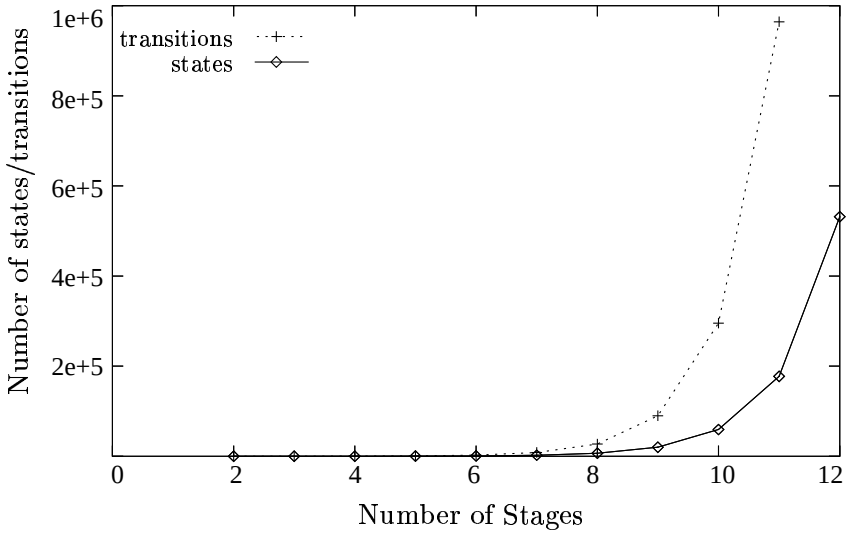

FIG. 6.1. *States and Transitions*

The time required to generate and solve the models must be carefully considered. The generation is always very quick: it takes less than 0.01 seconds to generate 20 models. The time required to solve the models is usually more important, especially when the models have a large state space. However, if we consider only relatively small models (up to 20,000 states), the resolution with the PEPA workbench takes only a few seconds. Fig. 6.1 shows that when the number of stages is less than 9, the size of the model is small enough to have a fast resolution. However, the model grows exponentially when the number of stages is increased, making AMoGeT less effective for a large number of stages. Since real applications usually do not have very many stages, this is not a limitation of the tool in practice.

The overall use of AMoGeT takes usually less than one minute for complex applications running on several processors, even when we consider several models to solve.

As stated earlier, in a scenario of long computing grid applications, with eventually dynamic rescheduling of the application, we consider that the tool may be run once per hour. We therefore believe that the amount of time required may be quite negligible and that the gain obtained by using the best of the mappings which were investigated can outperform the cost of the use of the tool.

7. Conclusions. In the context of grid applications, the availability and performance of the resources change dynamically. We have shown through this study that the use of skeletons, and performance models of these, can produce some relevant information to improve the performance of the application. This has been illustrated on the pipeline skeleton, which is a commonly used algorithmic skeleton. The models help us to choose the mapping, of the stages onto the processors, which will produce the best throughput. A tool automates all the steps to obtain the result easily.

The pipeline skeleton is a simple control skeleton. The deal skeleton has already been modelled in a similar way [3], and experiments are ongoing using deal skeletons

nested into a pipeline application. This approach will also be developed on some other skeletons so it may be useful for a larger class of applications.

Our recent work considers the generation of models which take into account information from the Grid resources, which is gathered with the help of the Network Weather Service [17]. This will allow us to have models fitted to the real-time conditions of the resources. This first case study has already shown that we can use such information productively and that we have the potential to enhance the performance of grid applications with the use of skeletons and process algebras.

Having process algebra models of our skeletons also potentially offers other benefits such as the ability to formally verify the correct functioning of the skeleton. We intend to explore this aspect in future work.

Appendix A. Structured Operational Semantics for PEPA.

The semantic rules, in the structured operational style, are presented in Figure A.1; the interested reader is referred to [13] for more details. The rules are read as follows: if the transition(s) above the inference line can be inferred, then we can infer the transition below the line. The notation $r_\alpha(E)$ which is used in the third cooperation rule denotes the *apparent rate* of α in E, i.e. the sum of the rates of all activities of type α in $\mathcal{A}ct(E)$.

Appendix B. Pipeline example: input file for the PEPA Workbench.

The input file for the PEPA Workbench is displayed in Fig. B.1, for a small example with $N_s = N_p = 3$, and where each processor is hosting one of the stages.

Appendix C. The PEPA Workbench.

C.1. Functioning of the Workbench. The PEPA Workbench begins by generating the reachable state space of a PEPA model as found from all possible interleavings of its transitions from state to state. For a finite state model with n states we can enumerate this state space as $C = \{C_1, \ldots, C_n\}$. As the workbench carries out this task it compiles the *infinitesimal generator matrix* Q of the continuous-time Markov process underlying the PEPA model. The workbench adds a transition rate r to Q_{ij} every time that it finds a transition from state C_i to C_j at rate r. Additionally it subtracts r from Q_{ii} in order that the row sum of the matrix remains in balance.

The conditions which must be satisfied in order to guarantee the existence of an equilibrium distribution for a Markov process, and for this to be the same as the limiting distribution, are well-known—*a* stationary *or* equilibrium *probability distribution,* $\mathbf{\Pi}$*, exists for every time-homogeneous irreducible Markov chain whose states are all positive-recurrent.*

The intuition behind this distribution is the obvious one, namely that in the long run the probability that the PEPA model is in state C_i is given by $\mathbf{\Pi}(C_i)$.

For finite state PEPA models whose derivation graph is strongly connected, and which therefore have generated an ergodic Markov process, the equilibrium distribution of the model, $\mathbf{\Pi}$, is found by solving the matrix equation

$$\mathbf{\Pi} Q = \mathbf{0} \tag{C.1}$$

subject to the normalisation condition which ensures that $\mathbf{\Pi}$ is a well-formed probability distribution

$$\sum \mathbf{\Pi}(C_i) = 1. \tag{C.2}$$

Prefix

$$\frac{}{(\alpha, r).E \xrightarrow{(\alpha,r)} E}$$

Cooperation

$$\frac{E \xrightarrow{(\alpha,r)} E'}{E \underset{L}{\bowtie} F \xrightarrow{(\alpha,r)} E' \underset{L}{\bowtie} F} (\alpha \notin L) \qquad \frac{F \xrightarrow{(\alpha,r)} F'}{E \underset{L}{\bowtie} F \xrightarrow{(\alpha,r)} E \underset{L}{\bowtie} F'} (\alpha \notin L)$$

$$\frac{E \xrightarrow{(\alpha,r_1)} E' \quad F \xrightarrow{(\alpha,r_2)} F'}{E \underset{L}{\bowtie} F \xrightarrow{(\alpha,R)} E' \underset{L}{\bowtie} F'} (\alpha \in L) \quad \text{where } R = \frac{r_1}{r_\alpha(E)} \frac{r_2}{r_\alpha(F)} \min(r_\alpha(E), r_\alpha(F))$$

Choice

$$\frac{E \xrightarrow{(\alpha,r)} E'}{E + F \xrightarrow{(\alpha,r)} E'} \qquad \frac{F \xrightarrow{(\alpha,r)} F'}{E + F \xrightarrow{(\alpha,r)} F'}$$

Hiding

$$\frac{E \xrightarrow{(\alpha,r)} E'}{E/L \xrightarrow{(\alpha,r)} E'/L} (\alpha \notin L) \qquad \frac{E \xrightarrow{(\alpha,r)} E'}{E/L \xrightarrow{(\tau,r)} E'/L} (\alpha \in L)$$

Constant

$$\frac{E \xrightarrow{(\alpha,r)} E'}{A \xrightarrow{(\alpha,r)} E'} (A \stackrel{def}{=} E)$$

FIG. A.1. *The operational semantics of PEPA*

The equations C.1 and C.2 are combined by replacing a column of **Q** by a column of ones and placing a 1 in the corresponding row of **0**.

Because the connectivity graph of the state transition system of the model will in general have low degree, the transition matrix of the Markov process is best stored as a sparse matrix. The PEPA Workbench uses a Java implementation of the pre-conditioned biconjugate gradient method. This is an iterative procedure as described in [15] storing the infinitesimal generator matrix in *row-indexed sparse storage mode*, a compact storage mode which requires storage of only about two times the number of nonzero matrix elements. An advantage of conjugate gradient methods for large sparse systems is that they reference the matrix only through its multiplication of a vector, or the multiplication of its transpose and a vector.

```
// PIPELINE APPLICATION
// 3 stages, 3 processors (1 stage per processor)

// Variables declaration (all identical)
mu1=10; mu2=10; mu3=10;
la1=10; la2=10; la3=10; la4=10;

// Definition of the Stages
Stage1 = (move1, infty).(process1, infty).(move2, infty).Stage1;
Stage2 = (move2, infty).(process2, infty).(move3, infty).Stage2;
Stage3 = (move3, infty).(process3, infty).(move4, infty).Stage3;

// Definition of the Processors
Processor1 = (process1, mu1).Processor1;
Processor2 = (process2, mu2).Processor2;
Processor3 = (process3, mu3).Processor3;

// Definition of the Network
Network = (move1,la1).Network + (move2,la2).Network
    + (move3,la3).Network + (move4,la4).Network;

// The pipeline model
Network <move1,move2,move3,move4>
    (Stage1 <move2> Stage2 <move3> Stage3)
    <process1,process2,process3> (Processor1||Processor2||Processor3)
```

FIG. B.1. *The input file for the PEPA Workbench:* `pipeline.pepa`

C.2. Computing performance results with the PEPA Workbench. The new functionality of the workbench is described through a tiny example [10], which we shall first describe. We then explain how to add the description of the results in the PEPA input file and how to compute them.

A tiny example. We describe the components of the PEPA input language for the Workbench via a simple example, described in the file `tiny.pepa`:

```
r1=2; r2=10; r3=1;
P1=(start,r1).P2;
P2=(run,r2).P3;
P3=(stop,r3).P1;
P1 || P1
```

This model is composed of two copies of a component, P_1, executing in a pure parallel synchronization. P_1 is a simple sequential process which undergoes a *start* activity with rate r_1 to become P_2 which runs with rate r_2 to become P_3 which goes back to P_1 via a *stop* activity with rate r_3.

The first line of the file is defining the rates. Then the sequential process is defined, and the final line is the *system equation*, which describes the behaviour of the modelled system.

Adding results to the input file. In order to automatically compute some performance results, the user just needs to specify them in the PEPA input file, for

example in the file `tiny.pepa` presented before. This is done by including at the end of the file one line per result, of the form:

```
result_name = {result_description};
result_name = rate * {result_description};
```

The name of the performance result that is described is `result_name`, and the description of the result for the PEPA State Finder is `result_description`.

The states of interest are described through the use of a simple pattern language, with double stars (`**`) for wild cards, and double vertical bars (`||`) for separators between model components. The model components are described in the order used in the system equation.

A `rate` can be added; in this case the final result obtained by the PEPA State Finder will be multiplied by this rate. This is quite useful to compute throughput.

For our example, we can add some results concerning the first process, independently of the state of the second one:

```
start1 = {P1 || **};
run1 = {P2 || **};
Trun1 = r2 * {P2 || **};
stop1 = {P3 || **};
```

For example, the performance result `run1` matches all the states in which the first process is ready to perform the *run* activity. The state of the second process can be anything. `Trun1` is the same, multiplied by the rate of the *run* activity `r2`. It corresponds therefore to the throughput of *run* for the first process.

For the pipeline application, the required performance result is specified in the PEPA input file `pipeline.pepa` (Fig. B.1). This is done by adding the following line at the end of this file:

```
Throughput = mu1 * { ** <move1,move2,move3,move4>
((process1, infty).(move2,infty).Stage1 <move2> ** <move3> **)
<process1,process2,process3> (** || ** || **)}
```

Computing the results. The results can be computed by using the command line interface. This is done by invoking the following command:

```
java pepa.workbench.Main -run lr ./tiny.pepa
```

The `-run lr` (or `-run lnbcg+results`) option means that we use the linear biconjugate gradient method to compute the steady state solution of the model described in the file `./tiny.pepa`, and then we compute the performance results specified in this file.

This execution prints the results to the screen, and it also saves one file per performance result (`./results/model_name.result_name`). This file is the output of the PEPA State Finder for the result description specified in the input file. It contains the state matching the description, and the sum of the steady-state probabilities for these states. It does not take the multiplicative rate into account. The results are also appended to the file `./model_root.res`, where `model_root` is the beginning of the `model_name`, until a "`_`" or a "`.`" is found. This is used to automatically compare results of similar models.

Only a few files have been modified to include the new functionality in the Java Workbench. The interested reader should refer to [12].

REFERENCES

[1] M. Alt, H. Bischof, and S. Gorlatch. Program Development for Computational Grids Using Skeletons and Performance Prediction. *Parallel Processing Letters*, 12(2):157–174, 2002.

[2] A. Benoit, M. Cole, S. Gilmore, and J. Hillston. Evaluating the performance of skeleton-based high level parallel programs. In M. Bubak, D. van Albada, P. Sloot, and J. Dongarra, editors, *The International Conference on Computational Science (ICCS 2004), Part III*, LNCS, pages 299–306. Springer Verlag, 2004.

[3] A. Benoit, M. Cole, S. Gilmore, and J. Hillston. Scheduling skeleton-based grid applications using PEPA and NWS. *Submitted to a special issue of The Computer Journal on Grid Performability Modelling and Measurement*, June 2004.

[4] R. Biswas, M. Frumkin, W. Smith, and R. Van der Wijngaart. Tools and Techniques for Measuring and Improving Grid Performance. In *Proc. of IWDC 2002 on Distributed Computing: Mobile and Wireless Computing*, volume 2571 of *LNCS*, pages 45–54, Calcutta, India, December 2002. Springer-Verlag.

[5] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press & Pitman, 1989. http://homepages.inf.ed.ac.uk/mic/Pubs/skeletonbook.ps.gz.

[6] M. Cole. eSkel: The **e**dinburgh **Ske**leton **l**ibrary. Tutorial Introduction. *Internal Paper, School of Informatics, University of Edinburgh*, 2002. http://homepages.inf.ed.ac.uk/mic/eSkel/.

[7] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004.

[8] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.

[9] N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field, and J. Darlington. ICENI: Optimisation of Component Applications within a Grid Environment. *Parallel Computing*, 28(12):1753–1772, 2002.

[10] S. Gilmore. The PEPA Workbench: User's Manual. *Internal Paper, School of Informatics, University of Edinburgh*, 2001. http://www.dcs.ed.ac.uk/pepa/pwb.pdf.

[11] S. Gilmore and J. Hillston. The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling. In *Proc. of the 7th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, number 794 in LNCS, pages 353–368, Vienna, May 1994. Springer-Verlag. http://www.dcs.ed.ac.uk/pepa/workbench.ps.gz.

[12] N.V. Haenel. User Guide for the Java Edition of the PEPA Workbench - Tabasco release. *Internal Paper, School of Informatics, University of Edinburgh*, 2003. http://www.dcs.ed.ac.uk/pepa/.

[13] J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

[14] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing (JPDC)*, 63(5):551–563, May 2003.

[15] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1992.

[16] F.A. Rabhi and S. Gorlatch. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer Verlag, 2002.

[17] R. Wolski, N.T. Spring, and J. Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6):757–768, 1999.