doi:10.1093/comjnl/bxh089

Scheduling Skeleton-Based Grid Applications Using PEPA and NWS

A. BENOIT, M. COLE, S. GILMORE AND J. HILLSTON

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

Any scheduling scheme for grid applications must make implicit or explicit assumptions about both the future behaviour of the application and the future availability and performance of grid resources. This paper describes an approach in which the future application behaviour is constrained by the use of algorithmic skeletons, facilitating modelling with a performance oriented process algebra, and future grid resource performance is predicted by the Network Weather Service (NWS) tool. The concept is illustrated through a case study involving Pipeline and Deal skeletons. A tool is presented which automatically generates and solves a set of models which are parameterised with information obtained from NWS. Some numerical results and timing information on the use of the tool are provided, illustrating the efficacy of this approach.

Handling Editor: Nigel Thomas

Received 29 June 2004; revised September 22, 2004

1. INTRODUCTION

Grid frameworks [1, 2] have enormous potential to provide significant quantities of computational power and storage for meeting the needs of today's most demanding computational tasks. Structured parallel programming languages are very valuable tools to deploy when programming in an environment such as a computational grid. The design of the language rules out problems such as deadlocks and process starvation which are faced by parallel application developers when working with low-level parallelism. One productive approach to high-level structured parallel programming is to use algorithmic skeletons [3] to structure the creation and configuration of processes. In this approach, the skeletons add expressive power to the programming language used for sequential computing blocks and expedite the development of complex parallel applications by providing generic and parametric parallel processing constructs to complement the loop constructs and conditional statements which are used in sequential computation.

Algorithmic skeletons successfully address the challenges of grid computation. In response to the changing workload on servers, or due to the sudden non-availability because of software or hardware problems, the application can be restructured to use an alternative implementation skeleton or there can be a simple re-evaluation of the parameters of the skeleton currently in use. Typically such resilience to operational faults is not found in low-level parallel programming methods and is one of the strengths of a structured approach to parallel programming. Furthermore, skeletons allow the programmer to provide explicit information about the future interaction structure of the application, which would be difficult or even impossible to derive statically from an equivalent unstructured program source.

The ability to reconfigure and redeploy an application on-the-fly would be used even more effectively if the future load on the available servers could be estimated with reasonable accuracy. The Network Weather Service (NWS) [4] provides us with these estimates. Analogous with meteorological climate prediction, a short-range forecast is made on the basis of a record of recent activity. Perfect forecasting of future loads is, of course, not possible, but in the experience of the users of the NWS, its predictions turn out more often useful than misleading so scheduling decisions based on NWS information can be very useful ones.

The detailed implementation and scheduling of the programmer selected skeletons is decided automatically according to the evaluation of performance models seeded with rate information obtained from the NWS. In this work, we use the algorithmic skeletons from Cole's eSkel library [5] and our performance models of these algorithmic skeletons are expressed in Hillston's Performance Evaluation Process Algebra (PEPA) [6]. A range of tools is available to solve PEPA models. We have extended and applied some of them in the present work.

In the next section, we present the principle of the Pipeline and Deal skeletons, which we use as a case study. A model of these skeletons is proposed in Section 3. In Section 4 we present a tool which automatically determines the best

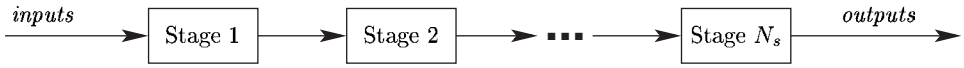

FIGURE 1. The pipeline application.

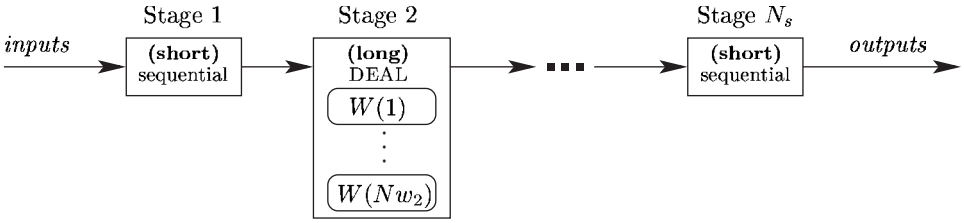

FIGURE 2. Pipeline and deal.

mapping to use for the application, by first generating a set of models, then solving them and comparing their results. Some numerical results are provided in Section 5. Finally, we give some conclusions.

2. PIPELINE AND DEAL SKELETONS

Many parallel algorithms can be characterized and classified by their adherence to one or more of a number of generic algorithmic patterns. A skeleton [3, 7, 8] is a programming construct which abstracts such a pattern of processes and interactions. The programmer invokes one or more skeletons to describe the structure of a program, specializing each with types and operations from the application domain. The code handling the interaction and invocation of the domain specific operations is inherited implicitly from the chosen skeleton. In this paper we review the concept of Pipeline parallelism, which has been successfully used in several applications, and of the Deal skeleton, which is often used nested inside a Pipeline skeleton.

2.1. Pipeline

In the simplest form of Pipeline parallelism [9], a sequence of N_s stages processes a sequence of *inputs* to produce a sequence of *outputs* (Figure 1). All inputs pass through each stage in the same order, with the processing of a particular input beginning as soon as its predecessor has left the first stage. It is noteworthy that parallelism is introduced by overlapping the processing of many input instances. Indeed, it is quite normal for the processing time of each individual input to be increased by pipelining. Performance benefits accrue when many such instances are processed concurrently across the pipeline.

2.2. Deal

It is evident that the performance of a pipeline is constrained by the processing time of its slowest stage. This may be improved by exploiting parallelism within, as well as between, stages. A simple approach is to implement the stage with several processors which take turns to accept inputs from the incoming stream, returning results to the output stream in the same order. In this manner, the slow processing time of all but the first input can be masked by this intra-stage parallelism (Figure 2). The Deal skeleton [5] abstracts this pattern. The name is chosen for its analogy with the process of dealing out a pack of cards (inputs) in strict rotation to a collection of players (stage internal processors). In our model there are Nw_s workers for a given Deal skeleton s.

From the application programmer's perspective, it is important to note that this approach is only valid for stateless stage computations. If the stage maintains and updates states from one input to the next, a more sophisticated internal parallelization strategy needs to be devised.

2.3. Context of the work

Considering both skeletons in the context of computational grids, we could map them to our computing resources. We have a set of processors, which may be different from one another, but are interconnected by an heterogeneous network.

As noted above, pipelining 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 the pipeline. Our work directly addresses the first of these issues by facilitating exploration of the stage-to-processor mapping space. It is the programmer's responsibility to tackle the second issue. Our approach assumes that the system will reach an equilibrium behaviour after running the application for a considerable time.

3. PEPA MODEL OF SKELETONS

In this section, we present our approach to modelling the Pipeline and Deal skeletons. The model is expressed in PEPA [6]. We first briefly introduce PEPA. The Pipeline and Deal skeletons can both be seen as particular cases of a Deal nested within a Pipeline application. We, therefore, present a generic model of Pipeline with Deal in Section 3.2.

3.1. 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. Time 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 used in the skeleton models, together with their names and interpretations, are presented 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 activities. The component $P + Q$ represents a system which may behave either as P or as Q—the activities of both are enabled. The first activity to be completed 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 explained by a defining equation.

Cooperation. In PEPA direct interaction, or cooperation, between components is the basis of compositionality. The set used as the subscript for the cooperation symbol, the cooperation set L, determines those activities on which the co-operands are forced to synchronize. 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 [6]). 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 synchronized in the final model.

The dynamic behaviour of a PEPA model is represented by the evolution of its components, as governed by the operational semantics of PEPA terms (see [6]). Thus, as in classical process algebra, the semantics of each term 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 and these form the nodes of the derivation graph which is formed by the exhaustive application of the semantic rules.

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 so as to 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 and labelling arcs connecting node P to node Q.

It is important to note that in our models the estimated duration of tasks, etc. is represented as random variables, not as constant values. These random variables are exponentially distributed. Repeated samples from the distribution will follow the distribution and conform to the mean but individual samples may potentially take any positive value.

3.2. Model of pipeline with deal

To model a Pipeline application we split the problem into the stages, the processors and the network. Some of the stages can then be modelled as Deal skeletons.

The stages. The first part of the model is the application model, which is independent of the resources on which the application will be computed. The application consists of N_s stages, which are each modelled by a PEPA component $Stage_s$ $(s = 1, \ldots, N_s)$.

When $Stage_s$ is not a Deal, it executes sequentially. As its first activity, it obtains data (activity $move_s$), then processes it (activity $process_{s,1}$) and finally moves the processed data to the next stage (activity $move_{s+1}$). In the $process_{s,1}$ activity, the 1 in the index denotes the first (and only) worker for this stage (i.e. the number of workers for the stage s is $Nw_s = 1$). The second subscript will play a more important role in Deal, when there may be many workers. The definition is in Figure 3a.

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

When $Stage_s$ is a Deal, we consider that we have Nw_s workers which have to process a sequence of inputs. In our model, we enforce cyclic allocation of inputs to workers by introducing, for each Deal, a *Source* component and a *Sink* component which interface between the Deal workers and the *move* actions which link this stage to its pipeline neighbours. Each worker $i \in \{1, \ldots, Nw_s\}$ first gets an input from the source with an $input_{s,i}$ action, processes it ($process_{s,i}$) then transfers its output to the sink ($output_{s,i}$).

We obtain the definitions $Source_s$, $Sink_s$ and $Worker_{s,i}$ from Figure 3b, where the workers are defined as $i = 1, \ldots, Nw_s$. All the workers are independent, and they

a. Stage without Deal

$$Stage_s \stackrel{def}{=} (move_s, \top).(process_{s,1}, \top).(move_{s+1}, \top).Stage_s$$

b. Stage with Deal

$$\begin{aligned}
Source_s &\stackrel{def}{=} (move_s, \top).(input_{s,1}, \top). \\
&\quad (move_s, \top).(input_{s,2}, \top). \\
&\quad \ldots \\
&\quad (move_s, \top).(input_{s,N_{ws}}, \top).Source_s \\
Worker_{s,i} &\stackrel{def}{=} (input_{s,i}, \top).(process_{s,i}, \top).(output_{s,i}, \top).Worker_{s,i} \\
Sink_s &\stackrel{def}{=} (output_{s,1}, \top).(move_{s+1}, \top). \\
&\quad (output_{s,2}, \top).(move_{s+1}, \top). \\
&\quad \ldots \\
&\quad (output_{s,N_{ws}}, \top).(move_{s+1}, \top).Sink_s \\
Stage_s &\stackrel{def}{=} Source_s \underset{LI_s}{\bowtie} \left(Worker_{s,1} \parallel \ldots \parallel Worker_{s,N_{ws}}\right) \underset{LO_s}{\bowtie} Sink_s
\end{aligned}$$

c. The Pipeline application

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

d. The processors

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

e. The network

$$\begin{aligned}
Network \stackrel{def}{=} & \quad (move_1, \lambda_1).Network & + \ldots + & \quad (move_{N_s+1}, \lambda_{N_s+1}).Network \\
+ & \quad (input_{s,1}, \lambda I_{s,1}).Network & + \ldots + & \quad (input_{s,N_{ws}}, \lambda I_{s,N_{ws}}).Network \\
+ & \quad (output_{s,1}, \lambda O_{s,1}).Network & + \ldots + & \quad (output_{s,N_{ws}}, \lambda O_{s,N_{ws}}).Network
\end{aligned}$$

f. Overall model

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

FIGURE 3. PEPA definitions.

are synchronized with the source and the sink via the *input* and *output* actions. We define $LI_s = \{input_{s,i}\}_{i \in \{1, \ldots, Nw_s\}}$ and $LO_s = \{output_{s,i}\}_{i \in \{1, \ldots, Nw_s\}}$ in the $Stage_s$ definition (Figure 3b).

Once all the stages have been defined, the Pipeline application is then a cooperation of the different stages over the $move_s$ activities, for $s = 2..N_s$. The activities $move_1$ and $move_{N_s+1}$ represent the arrival of an input into the application and the transfer of the final output out of the Pipeline, respectively. They do not represent any data transfer between stages, so they are not synchronized within the Pipeline application. As mentioned above, the rates on the input and output actions are left unspecified. These will be defined elsewhere in the model. The Pipeline definition is in Figure 3c.

The processors. We consider that the application must be mapped to a set of N_p processors. Each worker is implemented by a given (unique) processor, but a processor may host several workers. In order to keep the model simple, we put information about the processor (such as the load of the processor or the number of stages being processed) directly into the rate $\mu_{s,i}$ of the activities $process_{s,i}$, for $s = 1..N_s$ and $i = 1..N_{w_s}$ (these activities have been defined for the components $Stage_s$). Each processor is then represented by a PEPA component which has a cyclic behaviour, consisting of sequentially processing inputs for a worker. Some examples follow.

- In the case with no Deal, when $N_p = N_s$, we map one worker per processor:

$$Processor_i \stackrel{def}{=} (process_{i,1}, \mu_{i,1}).\ Processor_i$$

- If several workers are hosted by the same processor, we use a choice composition. In the following example ($N_p = 2$, $N_s = 2$, and the first stage is a Deal with

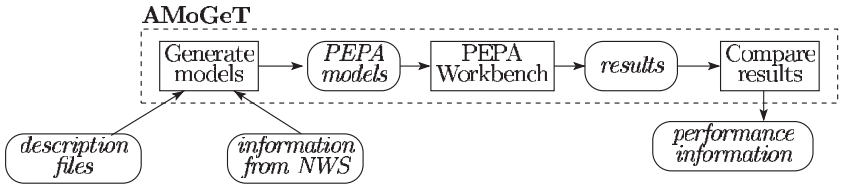

FIGURE 4. Principle of AMoGeT.

2 workers), the first processor processes the first worker of both stages and the second processor processes only the second worker of Stage 1 (so that Stage 1 is distributed across two processors).

$$Processor_1 \stackrel{def}{=} (process_{1,1}, \mu_{1,1}).\ Processor_1 + (process_{2,1}, \mu_{2,1}).\ Processor_1$$

$$Processor_2 \stackrel{def}{=} (process_{1,2}, \mu_{1,2}).\ Processor_2$$

Generally, since all processors are independent, the set of processors is defined as a parallel composition of the processor components (Figure 3d).

The network. Rather than directly representing the physical structure of the underlying network architecture, our network model is designed to allow us to derive the rates of the logical communication actions (*move*, *input*, *output*) of our Pipeline and Deal models from the NWS monitored physical processor to processor latency information. Using λ_s for the rate of a $move_s$ and $\lambda I_{s,i}$ and $\lambda O_{s,i}$ for the respective rates of $input_{s,i}$ and $output_{s,i}$ activities, the definition of the network is straightforward.

For example, assuming that only stage s is a Deal, we obtain the definition from Figure 3e. If some other stages are also modelled as Deals, we need to add their *input* and *output* activities into the choice. The relationship between NWS monitored information and the model's rates is complex and will be discussed in Section 4.4.

Pipeline with Deal. Once the different components of the model have been defined, we just have to map the stages onto the processors and the network by using the cooperation combinator. Two cooperation sets are used. L_n synchronizes the application and the network (it is the set of all the *move*, *input* and *output* activities), while L_p synchronizes the application and the processors (it is the set of all the *process* activities). The definition is in Figure 3f.

4. AMOGET: THE AUTOMATIC MODEL GENERATION TOOL

In this section we present the tool AMoGeT, which automatically generates and solves performance models for the Pipeline with Deal case study. This provides information which would allow the running grid application to be profitably rescheduled.

We start with an overview of the tool, then describe the inputs required and the principal functions of AMoGeT (model generation, model solution and results comparison).

4.1. Overall description of AMoGeT

Figure 4 illustrates the principle of the tool. In its current form, the tool is a generic analysis component. Its ultimate role will be that of 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. The tool allows everything to be done in a single step through a simple Perl script. On initialization, it obtains information from the resources and the network with the help of the NWS [4]. Some additional information must be provided to the tool via some *description files* (see below). The models are then generated and solved. Finally, the tool compares the results.

4.2. Description files for AMoGeT

We must initially give the tool the names of the processors which are to be used for the application. For this, we provide a file named `hosts.txt` containing a list of the IP addresses of the available computing resources. NWS must run on each of these nodes, and secure shell access must be allowed to gather some information about the processors. The first processor on the list is called the reference processor. All these processors are denoted in the following by processor i, where i is their ranking in the list, starting at 1 (the reference processor is processor 1).

Another file describes the modelled application. It is named `mymodel.des`, where mymodel is the name of the application. For a Pipeline skeleton, some information is needed on the stages of the Pipeline: the number of stages N_s, and the average time tr_s, in seconds, required to compute one output for stage s ($s = 1, \ldots, N_s$) on the reference processor:

$$\texttt{nbstage} = N_s; \texttt{tr1} = 10; \texttt{tr2} = 2; \ldots$$

The Deal skeleton can be defined in the same way, being in effect a Pipeline with only one stage. The time is then the time required to complete the work on the reference processor.

We also need to specify the size of the data transferred to and from each stage, in bytes. For $s = 1, \ldots, N_s + 1$, ds_s is the size of the data transferred to stage s, with the boundary case ds_{N_s+1} which represents the size of the output data:

$$\texttt{ds1} = 100; \texttt{ds2} = 5; \ldots$$

Finally we define a set of candidate mappings of stages and workers 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 $(1, 1, 2)$ indicates that the two first

stages are on processor 1, with the third stage on processor 2. When a Deal is nested inside a particular stage, we should indicate between brackets the processor of each worker. For a 2-stage Pipeline, the tuple $(1,\ (1, 2, 3))$ implies that the first stage is on processor 1, and that 3 workers are processing stage 2 on processors 1, 2 and 3, respectively. A mapping is then of the form $[input, tuple, output]$. The mapping definition is a set of mappings, it can be as follows:

$$\texttt{mappings} = [1, (1, 2, 3), 3], [1, (1, 2, (1, 2)), 2];$$

In this example, the first mapping is a standard Pipeline and the second one is a Pipeline with a Deal (2 workers) nested into the third stage.

4.3. Gathering information with the NWS

The NWS is a distributed system that periodically monitors and dynamically forecasts the performance that various network and computational resources can deliver over a given time interval. The service operates a distributed set of performance sensors (network monitors, CPU monitors, etc.) from which it gathers readings of the instantaneous conditions. It then uses numerical models to generate forecasts of what the conditions will be for a given time frame [4]. To run NWS, we only need to run a few scripts on the nodes that we want to monitor. The required information is obtained with the help of NWS sensors, which gather and store (time stamp, performance measurements) pairs for a specific resource. We can then obtain, in a direct manner, information about the fraction of CPU available to a newly-started process on each host, and the amount of time, in milliseconds, required to transmit a TCP message between two hosts. We denote by av_i the fraction of CPU available on the processor i, and by $\text{la}_{i,j}$ the latency of a communication from processor i to processor j (ms) for a message size of 1 byte.

Some additional information is required to evaluate the computing power of each processor. For this, we obtain the frequency of each CPU in MHz by reading this information from the file `/proc/cpuinfo`, using a secure shell connection to each host (thus limiting our work to Linux systems for the moment). This information is denoted cpu_i for processor i. This approach is somewhat naive since the performance of the processors depends on many other factors including memory access speed and cache policy. Moreover, it depends on the characteristics of the application. In future we plan to make a proper estimation of the performance of each processor for a given application, by running probes derived from the application. However, using the frequency of the processor gives us a rough idea of its global performance.

4.4. Generating the models

One model is generated from each mapping of the description file. Each model is as described in Section 3.2. The difficult point consists in generating the rates from the information gathered before. The model generation itself is then straightforward.

Rates μ. To compute the μ rates, we need to know how many workers are hosted on each processor, assuming that the work sharing between them is equitable. For a given stage s ($s = 1, \ldots, N_s$) and worker i ($i = 1, \ldots, Nw_s$), let j be the number of the processor hosting the worker i of the stage s. nb_j is the number of workers being processed on processor j. Then the rate associated with the $process_{s,i}$ activity is:

$$\mu_{s,i} = \frac{\text{av}_j}{\text{nb}_j} \times \frac{\text{cpu}_j}{\text{cpu}_1} \times \frac{1}{\text{tr}_s}$$

In effect, the available computing power av_j is further diluted by our own internal timesharing factor nb_j. The fraction $\text{cpu}_j/\text{cpu}_1$ represents the difference of computational power between the actual processor j and the reference processor 1, since the reference timing of the stage tr_s is done on processor 1.

Rates λ. The rates of communication (the variously subscripted λ, λI and λO terms) are derived from the processor to processor latency values (la) obtained from NWS and from the data size values (ds) of the data transferred from one stage to another. However, some of the *move*, *input* and *output* activities may express a data transfer from processor j to this same processor ($j = 1, \ldots, N_p$). The latencies $\text{la}_{j,j}$ must, therefore, be defined since no value is obtained for them from NWS. In order to work with tractable performance models, we treat such communication latency as insignificant. These activities are needed to ensure the logical behaviour of the model. They cannot be immediate because all activities are timed in PEPA. We set all these latencies to an arbitrarily small value of 10^{-5} ms. There are then three cases to consider.

A: Pipeline with no Deals. This is straightforward. Communication performance between stage $s-1$ and stage s is governed by the latency between the processor running stage $s-1$, which we label j_{s-1}, and the processor running stage s, labelled j_s. It depends also on the size of the data transiting between the two stages, which is ds_s. In this case there are no *input* and *output* actions (and therefore no λI and λO), so we must associate this communication cost with the *move* action. Since la_{j_{s-1},j_s} is in milliseconds we define

$$\lambda_s = \frac{10^3}{\text{ds}_s \times \text{la}_{j_{s-1},j_s}}$$

B: Pipeline with non-adjacent Deals. Suppose stage s is a Deal, but stages $s-1$ and $s+1$ are either simple stages or input or output sources. Additionally, suppose stage $s-1$ is mapped to processor j_{s-1}, stage $s+1$ is mapped to processor j_{s+1} and that worker i of the Deal ($i = 1, \ldots, Nw_s$) is mapped to processor $j_{s,i}$. In our model the communication of data between j_{s-1} and $j_{s,i}$ is captured by action $move_s$ followed by action $input_{s,i}$, with rates λ_s and $\lambda I_{s,i}$. The NWS monitored time corresponding to this is $\text{la}_{j_{s-1},j_{s,i}}$. Since λ_s is common to all such communications but $\lambda I_{s,i}$ is worker specific, we would like to allocate the full monitored rate to

$\lambda I_{s,i}$ as $10^3/(\mathrm{ds}_s \times \mathrm{la}_{j_{s-1},j_{s,i}})$, with λ_s infinite. However, since all activities are timed in PEPA models, we have to give a finite value to λ_s. We set it very high (arbitrarily set to 10^9, corresponding to a latency of 10^{-6}) and compensate the time spent by the $move_s$ activity in the definition of $\lambda I_{s,i}$ by removing 10^{-6} from the latency.

$$\lambda_s = 10^9$$
$$\lambda I_{s,i} = \frac{10^3}{\mathrm{ds}_s \times \mathrm{la}_{j_{s-1},j_{s,i}} - 10^{-6}}$$

The situation for output is symmetric, using this time the data size ds_{s+1}, and so

$$\lambda_{s+1} = 10^9$$
$$\lambda O_{s,i} = \frac{10^3}{\mathrm{ds}_{s+1} \times \mathrm{la}_{j_{s,i},j_{s+1}} - 10^{-6}}$$

C: Pipeline with adjacent Deals. Suppose that stages s and $s+1$ are both Deals. We need only consider the interfaces between these, since interaction with stages $s-1$ and $s+2$ will be handled either similarly, if these are also Deals, or by case B. Thus, we need to define λ_{s+1}, $\lambda O_{s,i}$ and $\lambda I_{s+1,i'}$ for $i = 1, \ldots, Nw_s$ and $i' = 1, \ldots, Nw_{s+1}$. As indicated above, we choose to associate all communication costs with *input* and *output* actions, so we set $\lambda_{s+1} = 10^9$.

In this paper, we choose to model an implementation of adjacent Deals in which communication between worker i (of the first Deal) and i' (of the second Deal) is achieved by sending the data via processor 1 of the first Deal. This simplification allows processor 1 of the first deal to reconcile the cyclic (by processor) output of the first Deal with the cyclic input of the second Deal. We expect to optimize away this extra communication in subsequent work. Thus, we define

$$\lambda O_{s,i} = \frac{10^3}{\mathrm{ds}_s \times \mathrm{la}_{j_{s,i},j_{s,1}}}$$
$$\lambda I_{s+1,i'} = \frac{10^3}{\mathrm{ds}_s \times \mathrm{la}_{j_{s,1},j_{s+1,i'}} - 10^{-6}}$$

Notice that we (correctly) compensate for the λ_{s+1} only once.

4.5. Solving the models

Numerical results have been computed from such models with the Java Version of the PEPA Workbench [10, 11]. The performance result that is pertinent to us is the throughput of the $move_s$ activities ($s = 1, \ldots, N_s + 1$), which represents the throughput of the Pipeline application. Since data passes sequentially through each stage, the throughput is identical for all s and we need to compute only the throughput of $move_1$ to obtain significant results. This is done by adding the steady-state probabilities of each state in which $move_1$ can happen, and multiplying this by λ_1. The result can be computed by a single command line, given that the result required is specified in the PEPA input file. This is done during the model generation (Section 4.4) by adding a line at the end of this file.

For a small example with two stages, two processors and no Deal, we have:

```
Thr = λ1 × {** || (Stage1 || **) || (** || **)}
```

If Stage 1 is a Deal with two workers,

```
Thr = 2 × λ1 × { ** || Source1 || **}
```

The expression between the braces {} describes the states of interest (in this example, the states in which $move_1$ can happen) through the use of a simple language. The double stars (`**`) are wild cards and the double vertical bars (`||`) are separators between model components. The model components are described in the same order used in the system equation and we can skip them at the end of the equation if there are only wild cards (as done in the second example).

Thus, the result for one model can be computed by using the command line interface of the PEPA workbench by invoking the following command:

```
java pepa.workbench.Main -run lr
  ./mymodel-[mapping].pepa
```

The `-run lr` option means that we use the linear biconjugate gradient method to compute the steady state solution of the model described in the file `./mymodel-[mapping].pepa`, and then we compute the performance results specified in this file—in this case, the `Throughput` result.

When a large number of models need to be solved, we use task farming to solve them in a parallel scheme. The most lightly loaded processors are selected with the help of information gathered previously with NWS, and the jobs are distributed on these processors. The gain is almost equal to the number of processors used, even if a small overhead can be observed due to the time required to dispatch the models and to collect the results.

Note that the scheduling for the Task Farm (which is effectively a simple skeleton) is dynamically determined by the pattern itself. There is therefore no need to analyse the scheduling of this skeleton and its straightforward use (without AMoGeT) is not a limitation for us.

4.6. Comparing the results

During the resolution, all the results are saved in a single file, and the last step, 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 some numerical results obtained on small experiments. These allow us to estimate the size of the models for different kinds of applications. Some measurements of the time required to run the tool are presented and finally

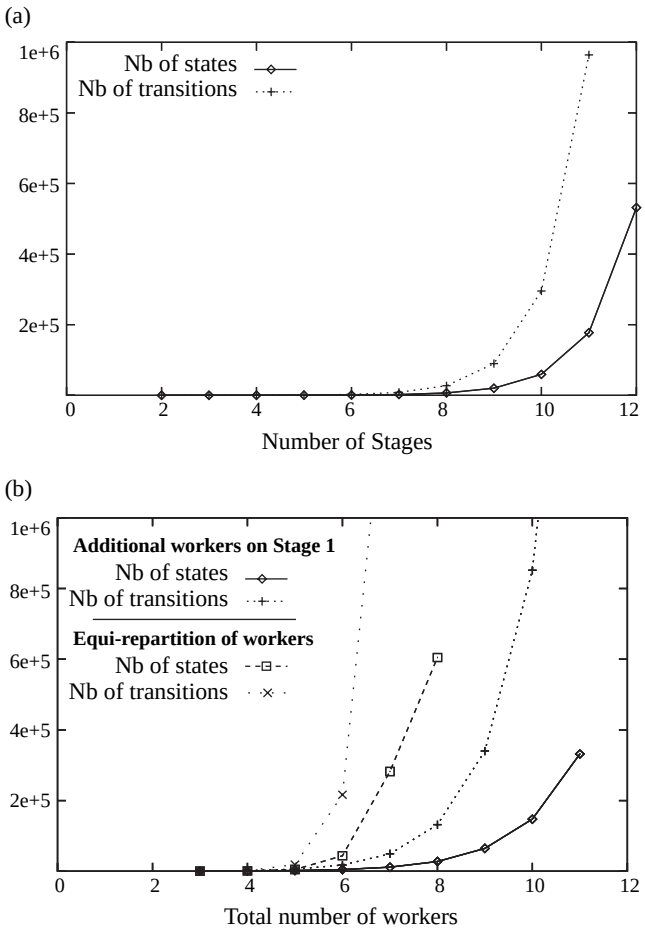

FIGURE 5. States and transitions. (a) Pipeline with no Deal-function of the number of stages. (b) Pipeline with Deal-3 Pipeline Stages (3 workers minimum).

we explain, through a few experiments, how information obtained by these techniques 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.

5.1. Size of the models

Figure 5 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 and Deal workers.

Figure 5a shows the case of a simple Pipeline without Deal. In this case we only consider the number of stages, since the total number of workers in the model equals the number of stages. When the number of stages is <9, the size of the model is such that the resolution is usually very quick (a few seconds). However, the model grows exponentially when the number of stages is increased, making AMoGeT less effective for a large number of stages. However, since real applications usually do not have very many stages, this is not a limitation of the tool in practice.

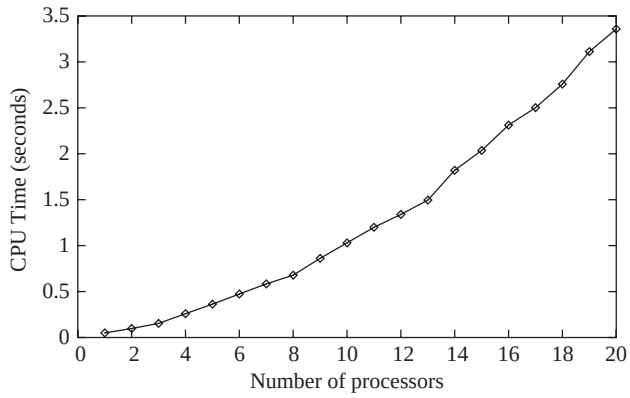

FIGURE 6. Time required to get information with NWS.

Figure 5b shows the impact of numerous workers on the size of the model. The study is limited to a Pipeline with 3 stages. We consider first the case where all additional workers are on the same stage, the first one, i.e. only the first stage is a Deal. In this case, the size of the model increases faster than for the Pipeline without Deal, and we can realistically consider only cases with <8 workers (6 workers for stage 1, and one for each other stages). When several stages are a Deal, and when we have approximately the same number of workers on each stage (equi-repartition of the workers), it is even worse and the models start to become very large with a total of only 6 workers.

5.2. Timing of the tool

The steps consuming time when using the AMoGeT tool are both the gathering of information from the NWS and the time required to generate and solve the models.

First, the time consumed by the use of NWS depends directly on the number of processors considered. Figure 6 represents the CPU time (in seconds) used by AMoGeT to gather information from NWS, as a function of the number of processors. This time is the mean value of five measurements done on a cluster of PCs. Different results are obtained from each measurement, due to the changing performance of the resources, so the results are much more significant when iterated. The global tendency is that the time increases with the number of processors, which is quite logical since a higher number of processors implies that we have to perform more calls to NWS. The amount of time required is only a few seconds for up to 20 processors and since the grid performance will not change rapidly, in general, we assume that we should make these measurements only once per hour or so, and then run AMoGeT to find out if a better scheduling should be adopted for the application.

The time required to generate and solve the models must also be carefully considered. The generation is always very quick: it takes <0.01 s 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 (Figure 5 illustrates this number of states function of the characteristics of the application).

Some tests have been done with larger models requiring around 40 s to be solved (up to 50,000 states and 250,000 transitions). In this case, several models are solved in parallel using task farming (see Section 4.5). The use of several processors allows us to consider such models without losing too much time on solving models.

The use of AMoGeT takes usually less than one minute for complex applications running on several processors, even when we consider several models which can be relatively large. The distributed resolution of the models allows us to decrease this time significantly. Considering that the tool may be run once an hour, it is likely that the amount of time required may be quite negligible and that the gain obtained by using the optimal scheduling can outperform the cost of the use of the tool, when we consider large applications with long stages.

5.3. Optimizing the application

We have made some experiments to study the performance improvement running an application using the best mapping obtained with AMoGeT in comparison to other mappings. For this, we compare the throughput obtained with the different models. Several experiments follow.

Experiment 1. In this first experiment, all processors are identical, and the four stages of the Pipeline execute in a time which is exponentially distributed with a mean of 10 s ($\mathrm{tr}_s = 10$, $s = 1..4$). Moreover, the network is homogeneous; we consider differing latencies to go from one processor to another but assume it is the same for the whole network, and we fix the size of the data transiting from one stage to another to 1. In this case we use a simple Pipeline without Deal, since all of the stages have the same complexity. The input and output data are located on processor 1.

Figure 7 shows the throughput as a function of the latency, for different mappings. For a small latency, the optimal mapping is obviously (1,2,3,4) (one stage per processor). When the latency is <1 s, the throughput for (1,2,3,4) is two times better than for (1,1,2,2) and four times better than for (1,1,1,1). The performance of the network has some impact on the optimal mapping only when the latency is of the same order as the time required to compute one stage ($\sim$10 s). Then it may be a better solution to share the stages between two processors, and finally just process everything on processor 1 when the network is exceptionally slow (latency >35 s). Notice also that for small latencies (<0.1 s), when the mean reference time to solve the stages is multiplied by a factor f, the resulting throughput is then divided by f.

For a latency of 0.01, the best throughput is obtained with the mapping (1,2,3,4) and it is 0.051. With the mapping (1,1,1,1), this throughput is almost four times

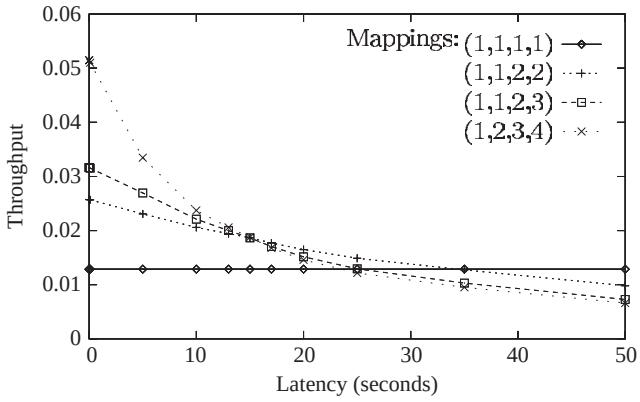

FIGURE 7. Experiment 1: throughput function of the latency, for different mappings.

worse (0.01287). Thus, a large number of inputs results in significant difference in the performance, and therefore better performance by choosing the optimal mapping. A less obvious case is for a latency of 20 s. The throughput for the mapping (1,1,2,2) is of 0.01648 instead of 0.01459 for the mapping (1,2,3,4). Even a reasonable number of inputs exhibit substantial benefits in choosing this mapping, since we save almost 8 s per input in this case.

Experiment 2. This second experiment investigates the use of a Deal skeleton nested inside a Pipeline in different cases. The Pipeline has three stages, and the network is working well (the latency of the order of 1 ms to go from one node to another, as observed with the NWS on our network). The size of the data is fixed to 1. The mean reference time for the stages is of the order of 10 s and the network has, therefore, no influence on this experiment (cf. Experiment 1). The input and output data are located on processor 1 and all workers have their own processor.

Figure 8 presents the results for identical stages (all with a mean reference time of 10 s) and for the case when the second stage is three times longer than the others ($\mathrm{tr}_2 = 30$). In the results displayed in columns 'P_2 working', all processors are identical, while for the experiment of the columns 'P_2 broken', the second processor (processing the first worker of Stage 2) is twice as heavily loaded as the others. Some additional workers are added to process stage 2 as a Deal (the case with no additional workers corresponds to a Pipeline without Deal). Note also that the throughput results are expressed in minutes (throughput per second multiplied by 60) for easy reading.

Adding workers to a stage (Deal skeleton) always improves the throughput of the application, especially when the stage is longer than the others, or when one of the processors processing this particular stage is not fully available. When one of the processors is broken (P_2 broken), it is not worth using when there are at least 2 additional workers. In this case, the throughput is better just by removing this worker (equivalent to P_2 working minus one worker). A study of all models allows us to observe which mapping is the best in this case.

Number of additional workers for Stage 2	Identical stages		Stage 2 three times longer	
	P_2 working	P_2 broken	P_2 working	P_2 broken
0	3.3844	2.3639	1.7584	0.9613
1	4.6408	3.9562	2.7070	1.7958
2	4.9294	4.5522	3.2482	2.4331
3	5.1061	4.8793	3.6643	2.9509
4	5.2283	5.0821	3.9970	3.3775
5	5.3191	5.2196	4.2683	3.7325

FIGURE 8. Result table for experiment 2 (throughput per minute).

We have seen from these experiments that AMoGeT can help us choose the optimal mapping for a skeleton-based application. Even if the difference in throughput is small, the time saved for large applications can be really worthwhile. It is therefore worth spending some time to solve the models.

6. CONCLUSIONS AND FUTURE WORK

We have reviewed the principles of the Pipeline and Deal skeletons and have constructed models of these using the PEPA. We have developed a tool which generates and solves such models, thereby providing information which could be used to profitably schedule and reschedule grid applications programmed using our skeletons. The models are augmented with real-time information concerning performance of the available grid resources, gathered with the help of the NWS.

Some other recent work considers the use of skeleton programs within grid nodes to improve the quality of cost information [12]. Each server provides a simple function capturing the cost of the 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 of such 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.

The implementation of the Deal skeleton that we discuss in this paper is not necessarily the most efficient, because of the use of strict polling. If one of the workers is on a slow processor, the whole application slows down. We plan to model other skeletons so that we can use AMoGeT on a larger class of applications. We plan also to use this approach based on skeletons and process algebra on real applications to illustrate its practical advantages. For example, we could model an application performing Optical Character Recognition using Pipeline and Deal skeletons. Moreover, in the current system, the user needs to specify a list of mappings from which the tool will select the best one. It would be useful to automatically generate a set of possible mappings, and maybe make a pre-selection of the mappings which may produce 'good' results in order to simplify the user's task and to improve the performance of the tool. This will be considered in future work. We also intend to address the issues raised by the use of exponential models having memoryless properties.

We believe that this first case study has already shown that our approach can allow grid systems to obtain important information and that we have the potential to enhance the performance of grid applications through the use of skeletons and process algebras.

REFERENCES

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

[2] Berman, F., Fox, G. and Hey, A. J. G. (eds) (2003) *Grid Computing: Making the Global Infrastructure a Reality*. Wiley & Sons.

[3] Cole, M. (1989) *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press & Pitman. Available at http://homepages.inf.ed.ac.uk/mic/Pubs/pubs.html.

[4] Wolski, R., Spring, N. and Hayes, J. (1999) The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Gener. Comp. Sys.*, **15**, 757–768.

[5] Cole, M. (2003) *eSkel: The* **E***dinburgh* **Ske***leton* **l***ibrary Version 2.0—Draft API reference manual*. Internal Paper, School of Informatics, University of Edinburgh, UK. Available at http://homepages.inf.ed.ac.uk/mic/eSkel/.

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

[7] Rabhi, F. and Gorlatch, S. (2002) *Patterns and Skeletons for Parallel and Distributed Computing*. Springer Verlag.

[8] Cole, M. (2004) Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, **30**, 389–406.

[9] Cole, M. (2002) *eSkel: The* **E***dinburgh* **Ske***leton* **l***ibrary. Tutorial Introduction*. Internal Paper, School of Informatics, University of Edinburgh, UK. Available at http://homepages.inf.ed.ac.uk/mic/eSkel/.

[10] Gilmore, S. and Hillston, J. (1994) The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling. In *Proc. 7th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, Vienna, May 3–6, *LNCS* **794**, pp. 353–368. Springer-Verlag.

[11] Haenel, N. V. (2003) *User Guide for the Java Edition of the PEPA Workbench*. Internal Paper, School of Informatics, University of Edinburgh. Available at http://www.dcs.ed.ac.uk/pepa.

[12] Alt, M., Bischof, H. and Gorlatch, S. (2002) Program development for computational grids using skeletons and performance prediction. *Parallel Processing Letters*, **12**, 157–174.