Automatic mapping of ASSIST applications using process algebra

Marco Aldinucci
Inst. of Information Science and Technologies – National Research Council (ISTI–CNR)
Via Moruzzi 1, Pisa I-56100, Italy
aldinuc@di.unipi.it

and

Anne Benoit
School of Informatics, The University of Edinburgh
Mayfield Road, Edinburgh EH9 3JZ, Scotland, UK
abenoit1@inf.ed.ac.uk

ABSTRACT

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. However, the programming and optimisation burden of a low level approach to grid computing is clearly unacceptable for large scale, complex applications. The development of grid applications can be simplified by using high-level programming environments. In the present work, we address the problem of the mapping of a high-level grid application onto the computational resources. In order to optimise the mapping of the application, we propose to automatically generate performance models from the application using the process algebra PEPA. We target applications written with the high-level environment ASSIST, since the use of such a structured environment allows us to automate the study of the application more effectively. Our methodology is presented through an example of a classical Divide&Conquer algorithm, together with results which demonstrate the efficiency of this approach.

Keywords: high-level parallel programming; ASSIST environment; Performance Evaluation Process Algebra (PEPA); automatic model generation; Divide&Conquer.

1. Introduction

A grid system is a geographically distributed collection of possibly parallel, interconnected processing elements, which all run some form of common grid middleware (e.g. Globus services) [11]. The key idea behind grid-aware applications is to make use of the aggregate power of distributed resources, thus benefiting from a computing power that falls far beyond the current availability threshold in a single site. However, developing programs able to exploit this potential is highly programmer intensive. Programmers must design concurrent programs that can execute on large-scale platforms that cannot be assumed to be homogeneous, secure, reliable or centrally managed. They must then implement these programs correctly and efficiently. As a result, in order to build efficient grid-aware applications, programmers have to address the classical problems of parallel computing as well as grid-specific ones:

1. *Programming:* code all the program details, take care about concurrency exploitation, among the others: concurrent activities set up, mapping/scheduling, communication/synchronisation handling and data allocation;
2. *Mapping & Deploying:* deploy application processes accordingly to a suitable mapping onto grid platforms. These may be highly heterogeneous in architecture and performance. Moreover, they are organised in a cluster-of-clusters fashion, thus exhibiting different connectivity properties among each pairs of platforms.
3. *Dynamic environment:* manage resource unreliability and dynamic availability, network topology, latency and bandwidth unsteadiness.

Hence, the number and quality of problems to be resolved in order to draw a given QoS (in term of performance, robustness, etc.) from grid-aware applications is quite large. The lesson learnt from parallel computing suggests that any low-level approach to grid programming is likely to raise the programmer's burden to an unacceptable level for any real world application. Therefore, we envision a layered, high-level programming model for the grid, which is currently pursued by several research initiatives and programming environments, such as ASSIST [18], eSkel [7], GrADS [16], ProActive [4], Ibis [17]. In such an environment, most of the grid specific efforts are moved from programmers to grid tools and run-time systems. Thus, the programmers have only the responsibility of organising the application specific code, while the programming tools (i.e. the compiling tools and/or the run-time systems) deal with the interaction with the grid, through collective protocols and services [10].

In such a scenario, the QoS and performance constraints of the application can either be specified at compile time or varying at run-time. In both cases, the run-time system should actively operate in order to fulfil QoS requirements of the application, since any static resource assignment may violate QoS constraints due to the very uneven performance of grid resources over time. As an example, ASSIST applications exploit an autonomic (self-optimisation) behaviour. They may be equipped with a QoS contract describing the degree of performance the application is required to provide. The ASSIST run-time tries to keep the QoS contract valid for the duration of the application run despite possible variations of platforms performance at the level of grid fabric [3]. The autonomic features of an ASSIST application rely heavily on run-time application monitoring, and thus they are not fully effective for application deployment since the application is not yet running. In order to deploy an application onto the grid, a suitable mapping of application processes onto grid platforms should be established, and this process is quite critical for application performance.

In this paper we address this problem by defining a performance model of an ASSIST application in order to statically optimise the mapping of the application onto a heterogeneous environment. The model is generated from the source code of the application, before the initial mapping. It is expressed with the process algebra PEPA [13], designed for performance evaluation. The use of a stochastic model allows us to take into account aspects of uncertainty which are inherent to grid computing, and to use classical techniques of resolution based on Markov chains to obtain performance results. This static analysis of the application is complementary with the autonomic reconfiguration of ASSIST applications, which works on a dynamic basis. In this work we concentrate on the static part to optimise the mapping, while the dynamic management is done at run-time. It is thus an orthogonal but complementary approach.

Structure of the paper. The next section introduces the ASSIST high-level programming environment and its run-time. Section 3 introduces the Performance Evaluation Process Algebra PEPA, which is used to model ASSIST applications (Section 4). These performance models help to optimise the mapping of the application. We illustrate our approach on the Data Mining C4.5 algorithm, and performance results are provided to show the effectiveness of this approach. Finally, concluding remarks are given in Section 5, as well as future work directions.

2. The ASSIST environment and its run-time support

ASSIST (A Software System based on Integrated Skeleton Technology) is a programming environment aimed at the development of distributed high-performance applications [18,2]. ASSIST applications should be compiled in binary packages which can be deployed and run on grids, including those exhibiting heterogeneous platforms. Deployment and run is provided through standard middleware services (e.g. Globus) enriched with the ASSIST run-time support.

2.1. The ASSIST coordination language

ASSIST applications are described by means of a coordination language, which can express arbitrary graphs of modules, interconnected by typed streams of data. Each stream realises a one-way asynchronous channel between two sets of endpoint modules: sources and sinks. Data items injected from sources are broadcast to all sinks. All data items injected into a stream should match stream type.

Modules can be either sequential or parallel. A sequential module wraps a sequential function. A parallel module *(parmod)* can be used to describe the parallel execution of a number of sequential functions that are activated and run as *Virtual Processes* (VPs) on items arriving from input streams. The VPs may synchronise with the others through barriers. The sequential functions can be programmed by using a standard sequential language (C, C++, Fortran). A *parmod* may behave in a data-parallel (e.g. SPMD/for-all/apply-to-all) or task-parallel (e.g. farm) way and it may exploit a distributed shared state which survives to VPs lifespan. A module can nondeterministically accept from one or more input streams a number of input items, which may be decomposed in parts and used as function parameters to instantiate VPs according to the input and distribution rules specified in the parmod. The VPs may send items or parts of items onto the output streams, and these are gathered according to the output rules.

An ASSIST application is sketched in Appendix A. We briefly describe here how to code an ASSIST application and its modules; more details on the particular application in Appendix A are given in Section 4.1. In lines 5–6 four streams with type `task_t` are declared. Lines 7–10 define endpoints of streams. Overall, lines 4–11 define the application graph of modules. In lines 14–18 two sequential modules are declared: these simply provide a container for a sequential function invocation and the binding between streams and function parameters. In lines 21–56 two parmods are declared. Each parmod is characterised by its `topology`, `input_section`, `virtual_processes`, and `output_section` declarations.

The `topology` declaration specialises the behaviour of the Virtual Processes as farm (topology `none`, as in line 45), or SMPD (topology `array`). The `input_section` enables programmers to declare how VPs receive data items, or parts of items, from streams. A single data item may be distributed (scattered, broadcast or unicast) to many VPs. The `input_section` realises a CSP repetitive command [14]. The `virtual_processes` declarations enable the programmer to realise a parametric Virtual Process starting from a sequential function (`proc`). VPs may be identified by an index and may synchronise and exchange data one with another through the ASSIST language API. The `output_section` enables programmers to declare how data should be gathered from VPs to be sent onto output streams. More details on the ASSIST coordination language can be found in [18,2].

2.2. The ASSIST run-time support

The ASSIST compiler translates a graph of modules into a network of processes. As sketched in Fig. 1, sequential modules are translated into sequential processes, while parallel modules are translated into a parametric (w.r.t. the parallelism degree) network of processes: one *Input Section Manager* (ISM), one *Output Section Manager* (OSM), and a set of *Virtual Processes Managers* (VPMs, each of them running a set of Virtual Processes). The actual parallelism degree of a parmod instance is given by the number of VPMs. Also, a number of processes are devoted to application QoS control, e.g. a *Module Adaptation Manager* (MAM), and an *Application Manager* (AM) [3].

The processes which compose an ASSIST application communicate via ASSIST support channels. These can be implemented on top of a number of grid middleware communication mechanisms (e.g. shared memory, TCP/IP, globus, CORBA-IIOP, SOAP-WS). The suitable communication mechanism between each pair of processes is selected at launch time depending on the mapping of the processes.

2.3. Toward fully grid-aware applications

ASSIST applications can already cope with platform heterogeneity [1], either in space (various architectures) or in time (varying load) [3]. These are definite features of a grid, however they are not the only ones. Grids are usually organised in sites

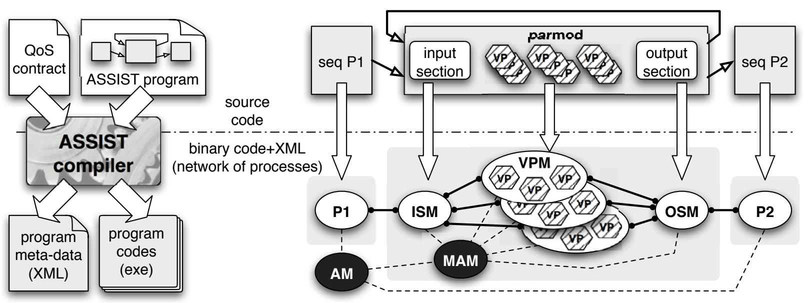

Fig. 1. An ASSIST application and a QoS contract are compiled in a set of executable codes and its meta-data [2]. This information is used to set up a processes network at launch time.

on which processing elements are forming a Virtual Private Network allowing only outbound connections. Also, they are often fed through job schedulers. In these cases, setting up a multi-site parallel application onto the grid is a challenge in its own right (irrespectively of its performance). Advanced reservation, co-allocation, multi-site launching are currently hot topics of research for a large part of the grid community. Nevertheless, many of these problems should be targeted at the middleware layer level and they are largely independent of the logical mapping of application processes on a suitable set of resources, given that the mapping is consistent with deployment constraints.

In this work, we assume that the middleware level supplies (or will supply) suitable services for co-allocation, staging and execution. These are actually the minimal requirements in order to imagine the bare existence of any non-trivial, multi-site parallel application. Thus our main concern is to analyse how we can map an ASSIST application assuming that we can exploit middleware tools to deploy and launch applications [9].

3. Introduction to performance evaluation and PEPA

In this section, we briefly introduce the Performance Evaluation Process Algebra PEPA [13], with which we plan to model the ASSIST application in order to help the mapping of the application. The use of a process algebra allows us to include the aspects of uncertainty relative to both the grid and the application, and to use standard methods to easily and quickly obtain performance results.

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 in order to help understanding the models in Section 4.

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 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.

Cooperation: In PEPA direct interaction, or *cooperation*, between components is the basis of compositionality. The component $P \underset{L}{\bowtie} Q$ represents a system in which components P and Q are forced to synchronise on a set of activities L. For action types not in L, the two components proceed independently and concurrently with their enabled activities. However, an activity whose action type is in L cannot proceed until both components enable an activity of that type. The two components then proceed together to complete the *shared activity.* We write $P \parallel Q$ as an abbreviation for $P \underset{L}{\bowtie} Q$ when L is empty. 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, as governed by the operational semantics of PEPA terms [13]. Thus, as in classical process algebra, the semantics of each term is given via a labelled *multi-transition* system (the multiplicity 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 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. Examples of derivation graphs can be found in [13].

It is important to note that in our models the rates are represented as random variables, not 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. The use of such distribution is quite realistic and it allows us to use standard methods on CTMCs to readily obtain performance results. There are indeed several methods and tools available for analysing PEPA models. Thus, the PEPA Workbench [12] allows us to generate the state space of a PEPA model and the infinitesimal generator matrix of the underlying Markov chain. The state space of the model is represented as a sparse matrix. The PEPA Workbench can then compute the steady-state probability distribution of the system, and performance measures such as throughput and utilisation can be directly computed from this.

4. Performance models of ASSIST application

Our main contribution in this paper is to present an approach to automatically generate PEPA models (c.f. Section 3) from an ASSIST application (c.f. Section 2), in order to optimise the static mapping of the application.

We first introduce an example of Data Mining classification algorithm (Section 4.1). The PEPA model of this application is presented in Section 4.2. We show in Section 4.3 how PEPA models can be derived automatically from the source code of the application. Finally, we explain how the performance results are obtained and we give some numerical results to illustrate how this information helps for an efficient mapping of an ASSIST application (Section 4.4).

4.1. The C4.5 algorithm

To introduce our methodology, we concentrate on an example already implemented in ASSIST, which is the data mining C4.5 classification algorithm [15]. Data mining consists of the non-trivial extraction of implicit, previously unknown information from data. In particular, C4.5 builds decision trees, i.e. a technique for building classification models for predicting classes for unseen records. These simple structures represent a classification (and regression) model. Starting at the root node, a simple question is asked (usually a test on a feature value, for example $Age < 35$). The branches emanating from the node correspond to alternative answers (e.g. Yes/No). Once a leaf node is reached (one from which no branches emanate) we take the decision or classification associated with that node.

Here, we outline only the information needed to model ASSIST-C4.5 with PEPA and to study its mapping onto grid. A full description of the parallel algorithm and ASSIST implementation can be found in [2,8]. The code of the application is sketched in Appendix A, and the line numbers in the following refer to this appendix. The structure of the application can be represented as a graph, where the modules are the nodes and the streams the arcs. The graph representing this application is displayed in Fig. 2 ➊. It can be automatically deduced from the `main` procedure of the ASSIST program (lines 4–11).

The C4.5 algorithm is designed according to the Divide&Conquer paradigm;

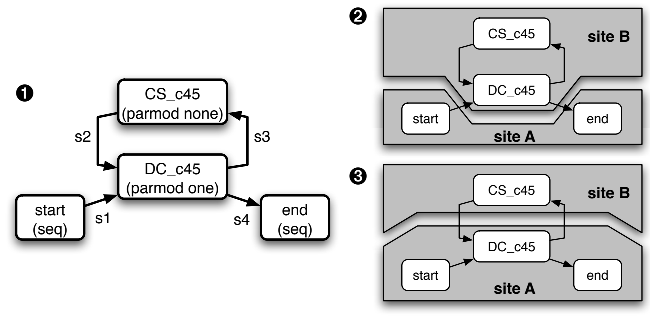

Fig. 2. Graph representation of the C4.5 application (❶) and two different multi-site deployments (❷, ❸).

all algorithms following the same paradigm could be similarly implemented. The C4.5 is implemented by means of four modules: the `start` module is generating the inputs, which are data sets or subsets, while the `end` module is collecting outputs (the data sets' classification). Modules `DC_c45` and `CS_c45` represent the core of the algorithm. The `DC_c45` drives both Divide and Conquer phases: it outputs data items which are obtained from the split (Divide) or join (Conquer) of input stream items. The `CS_c45` module behaves as "co-processor" of the first module: it receives a data item and sort it out in such a way that it can be split trivially: it chooses which attribute gives the better information gain when the data is classified with respect to it (by counting and sorting data). Since this forms most of the computational effort of the whole process, these computations are done in a parallel module. The result is sent back to `DC_c45`, and this module takes an item either from stream `s1` or from stream `s2` as an input. For the output policy, `DC_c45` either sends a task for computation to `CS_c45` or outputs a result to `end`. This behaviour can be observed from the guards in the input section of the code (disjoint guards, lines 24–29), and from the output section (lines 38–41). In all cases stream items carry references to data exploiting ASSIST shared memory to minimize data size transfers over channels.

4.2. PEPA models of ASSIST application

To model such an application, each module is represented as a PEPA component, and the different components are synchronised through the streams to model the overall application. We present in this section the PEPA model of the C4.5 application, and similar models can be obtained from any ASSIST applications. In the C4.5 algorithm, the module `DC_c45` can take as an input an item from either of the streams `s1` or `s2`. An output can be generated on either of the streams `s3` or `s4`. The user should provide some additional information to inform the model generator about the theoretical complexity of the modules (related to the amount of computations) and of the communications (size of the data on each stream, etc). This information is provided directly in the ASSIST source code. This will be de-

$$
\begin{array}{rcl}
S1 & \stackrel{\text{def}}{=} & S2 \\
S2 & \stackrel{\text{def}}{=} & (pS, \mu_S).S3 \\
S3 & \stackrel{\text{def}}{=} & (s1, \lambda_1).S1 \\
DC1 & \stackrel{\text{def}}{=} & (s1, \top).DC2 + (s2, \top).DC2 \\
DC2 & \stackrel{\text{def}}{=} & (pDC, \mu_{DC}).DC3 \\
DC3 & \stackrel{\text{def}}{=} & (s3, \lambda_3).DC1 + (s4, \lambda_4).DC1 \\
CS1 & \stackrel{\text{def}}{=} & (s3, \top).CS2 \\
CS2 & \stackrel{\text{def}}{=} & (pCS, \mu_{CS}).CS3 \\
CS3 & \stackrel{\text{def}}{=} & (s2, \lambda_2).CS1 \\
E1 & \stackrel{\text{def}}{=} & (s4, \top).E2 \\
E2 & \stackrel{\text{def}}{=} & (pE, \mu_E).E3 \\
E3 & \stackrel{\text{def}}{=} & E1
\end{array}
$$

Fig. 3. PEPA model of the example

tailed in Section 4.3, in order to show how this information is used to automatically generate the PEPA model.

The PEPA components of the modules are shown in Fig. 3. The activity `sX` ($X = 1, 2, 3$) represents the transfer of a data through the stream `X`, with the associated rate λ_X. The activity `pM` (M = `S (start), DC (DC_c45), CS (CS_c45), E (end)`) represents the processing of a data in module `M`, with a rate μ_M. The overall PEPA model is then obtained by a collaboration of the different modules in their initial states: $S1 \underset{s1}{\bowtie} DC1 \underset{s2,s3}{\bowtie} CS1 \underset{s4}{\bowtie} E1$. The performance results obtained are the probability to be in either of the states of the system. We compute the probability to be waiting for a processing activity `pM`, or to wait for a transfer activity `sX`. From this information, we can determine the bottleneck of the system and decide the best way to map the application onto the available resources.

4.3. Automatic PEPA model generation

To allow an automatic generation of PEPA models from the ASSIST source code, we ask the user to provide some information directly in the main procedure of the application. This information must specify the rates of the different activities of the PEPA model. We are interested in the relative computational and communication costs of the different parts of the system, but we define numerical values to allow a numerical resolution of the PEPA model. The complexity of the modules depends of the computations done, and also of the degree of parallelism used for a parallel module. In our application, the modules `start`, `end` and `DC_c45` are not computationally intensive, while `CS_c45` is doing the core of the computation. We therefore assign an average duration of 1 sec. to the last module, relatively to 0.01 sec. for the others.

In our example, we expect the amount of communication on each stream to be the same, since the same size of object (a reference to the real object) is transferred. The actual data is available in a shared memory, but this is beyond the scope of our PEPA model. We consider two different cases of application mapping, given that our environment consists in two clusters linked through a slow network (c.f. Fig. 2). In the first case (Fig. 2 ❷), we map the loop onto the same cluster, and the `start` and `end` modules onto another. This means that communications on streams s1 and s4 are slow (defined arbitrarily to 1 second), while they are fast on s2 and s3 (0.001 seconds). In the second case (Fig. 2 ❸), we split the loop into two parts, thus `start`, `end` and `DC_c45` are on the same cluster while the computational part `CS_c45` is on the other cluster by itself. In this case, communications on s1 and s4 are fast, while they are slow on s2 and s3.

This information is then defined in the source code by calling the `rate` function, in the body of the `main` procedure of the application (Appendix A, between lines 10 and 11). The rate function should be called for each stream and each module defined in order to fix the rates. We can define several sets of rates in order to compare several PEPA models. The values for each sets are defined between brackets, separated with commas, as shown in the example below.

```
rate(s1)=(1,1000); rate(s2)=(1000,1); rate(s3)=(1000,1);
rate(s4)=(1,1000); rate(start)=(100,100); rate(end)=(100,100);
rate(DC_c45)=(100,100); rate(CS_c45)=(1,1);
```

The PEPA model is generated during a precompilation of the source code of ASSIST. The parser identifies the `main` procedure and extracts the useful information from it: the modules and streams, the connections between them, and the rates of the different activities. The main difficulty consists in identifying the schemes of input and output behaviour in the case of several streams. This information can be found in the input and output section of the parmod code. Regarding the input section, the parser looks at the guards. Disjoint guards means that the module takes input from either of the streams when some data arrives. This is translated by a choice in the PEPA model, as illustrated in our example. However, some more complex behaviour may be expressed, as for instance saying that the parmod can start executing only when it has data from both streams. In this case, the PEPA model is changed with some sequential composition to express this behaviour. For example, $DC1 \overset{\text{def}}{=} (s1, \top).(s2, \top).DC2 + (s2, \top).(s1, \top).DC2$. Another problem may be introduced by variables in guards, since these may change the frequency of accessing data on a stream. Since the variables may depend on the input data, we cannot automatically extract static information from them. They are currently ignored, but we plan to address this problem by asking the programmer to provide the relative frequency of the guard. The considerations for the output section are similar.

The PEPA model generated by the C4.5 application for the first set of rates is represented in Fig. 4. For the second set of rates, the PEPA model is identical except from the rates $\mathtt{la}X$ $(X = 1..4)$ in the figure.

4.4. Performance results

Once the PEPA models have been generated, performance results can be obtained easily with the PEPA Workbench [12]. Some additional information is gen-

```
muS=100;muDC=100;muCS=1;muE=100;
la1=1;la2=1000;la3=1000;la4=1;

S1=S2; S2=(pS,muS).S3; S3=(s1,la1).S2;
DC1=(s1,infty).DC2 + (s2,infty).DC2; DC2=(pDC,muDC).DC3;
DC3=(s3,la3).DC1 + (s4,infty).DC1;
CS1=(s3,infty).CS2; CS2=(pCS,muCS).CS3; CS3=(s2,la2).CS1;
E1=(s4,la4).E2; E2=(pE,muE).E3; E3=E1;

(S1 <s1> (DC1 <s2,s3> CS1)) <s4> E1
```

Fig. 4. PEPA model code for the example: file example.pepa

erated in the PEPA source code of Fig. 4 to specify the performance results that we are interested in. This information is the following:

```
moduleS = 100 * {S2 || **  || **  || **}; moduleDC= 100 * {** || DC2 || **  || **};
moduleCS= 100 * {** || **  || CS2 || **}; moduleE = 100 * {** || **  || **  || E2};
stream1 = 100 * {S3 || DC1 || **  || **}; stream2 = 100 * {** || DC1 || CS3 || **};
stream3 = 100 * {** || DC3 || CS1 || **}; stream4 = 100 * {** || DC3 || **  || E1};
```

The expression in brackets describes the states of the PEPA model corresponding to a particular state of the system. For each module M, the result M corresponds to the percentage of time spent waiting to process this module. The steady-state probability is multiplied by 100 for readability and interpretation reasons. A similar result is obtained for each stream. We expect the complexity of the PEPA model to be quite simple and the resolution straightforward for most of the ASSIST applications. In our example, the PEPA model consists in 36 states and 80 transitions, and it requires less than 0.1 seconds to generate the state space of the model and to compute the steady state solution, using the linear biconjugate gradient method [12]. The results for the example are depicted in Fig. 1.

`Performance result`	`Percentage − case1`	`Percentage − case2`
moduleS	0.49	52.14
moduleDC	1.23	52.24
moduleCS	74.10	10.60
moduleE	0.49	52.14
$s1$	49.37	5.21
$s2$	0.07	10.61
$s3$	0.07	10.61
$s4$	49.37	5.21

Table 1. Performance results for the example

These two cases of study provide us information about the behaviour of the application when mapped in two different ways. In the first case (loop grouped on the same site), most of the time is spent computing the `CS_c45` module, which is the computational part of the application. The communication time is spent sending the data into the loop, on streams s1 and s4. In the second case (loop divided between two sites), we can see that a lot of time is spent in the non-computationally intensive modules, because `CS_c45` is continuously waiting for data. To optimise the application, it is thus better to map it as sketched in case 1.

The main aim of this analysis is to compare alternative mappings. In fact, communication and computation rates already include mapping peculiarities (speed of individual links and processors). With the same technique it is also possible to conduct a predictive analysis. Rates are assigned to the PEPA model solely on the basis of the application logical behaviour, assuming uniform speed of connections and processors. In this way the result of the analysis is not representing a particular mapping, it rather highlights individual resources (links and processors) requirements, that are used to label the application graph. On this basis, it is possible to approach the mapping problem as a weighted graph clusterisation problem. As an example, let us assume that the target grid has inter-site connections that are slower than intra-site ones. The goal of the clusterisation is to find a set of cuts in the graph which minimise the aggregate weight of cut-crossing edges (where the number of cuts depends on the required number of sites we would like to map on to), provided that the weight of the nodes assigned to a site is compatible with the power of this site platform.

5. Conclusions and perspectives

In this paper we have presented a new technique to automatically generate PEPA models from an ASSIST application with the aim of improving the mapping of the application. This is is an important problem in grid application optimisation. These models and the process of generating them have been detailed, together with examples of performance results. The C4.5 application has been chosen to explain our methodology. It is our belief that having an automated procedure to generate PEPA models and obtain performance information may significantly assist in taking mapping decisions. However, the impact of this mapping on the performance of the application with real code requires further experimental verification. This work is ongoing, and is coupled with further studies on more complex applications.

The approach described here considers the modules as blocks and does not model the internal behaviour of each module. A more sophisticated approach might be to consider using known models of individual modules and to integrate these with the global ASSIST model, thus providing a more accurate indication of the performance of the application. At this level of detail, distributed shared memory and external services (e.g. DB, storage services, etc) interactions can be taken into account and integrated to enrich the network of processes with dummy nodes representing external services. PEPA models have already been developed for pipeline or deal skeletons [5,6], and we could integrate such models when the parmod module has been adapted to follow such a pattern.

Analysis precision can be improved by taking into account historical (past runs) or synthetic (benchmark) performance data of individual modules and their communications. This kind of information should be scaled with respect to the expected performances of fabric resources (platform and network performances), which can be retrieved via the middleware information system (e.g. Globus GIS).

References

[1] M. Aldinucci, S. Campa, M. Coppola, S. Magini, P. Pesciullesi, L. Potiti, R. Ravazzolo, M. Torquati, and C. Zoccolo. Targeting heterogeneous architectures in ASSIST: experimental results. In *10th Intl Euro-Par 2004*, volume 3149 of *LNCS*, pages 638–643, Pisa, Italy, Aug. 2004. Springer.

[2] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, and C. Zoccolo. ASSIST as a research framework for high-performance Grid programming environments. In *Grid Computing: Software environments and Tools*. Springer, 2005. (To appear, draft available as TR-04-09, Uni. Pisa, Italy, Feb. 2004).

[3] M. Aldinucci, A. Petrocelli, E. Pistoletti, M. Torquati, M. Vanneschi, L. Veraldi, and C. Zoccolo. Dynamic reconfiguration of grid-aware applications in ASSIST. In *11th Intl Euro-Par 2005*, LNCS, Lisboa, Portugal, Aug. 2005. Springer. To appear.

[4] F. Baude, D. Caromel, and M. Morel. On hierarchical, parallel and distributed components for Grid programming. In *Workshop on component Models and Systems for Grid Applications*, ICS '04, Saint-Malo, France, June 2005.

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

[6] A. Benoit, M. Cole, S. Gilmore, and J. Hillston. Scheduling skeleton-based grid applications using PEPA and NWS. *The Computer Journal*, 48(3):369–378, 2005.

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

[8] M. Coppola and M. Vanneschi. Parallel and Distributed Data Mining through Parallel Skeletons and Distributed Objects. In *Data Mining: Opportunities and Challenges*, pages 106–141. IDEA Group Publishing, 2003.

[9] M. Danelutto, M. Vanneschi, C. Zoccolo, N. Tonellotto, R. Baraglia, T. Fagni, D. Laforenza, and A. Paccosi. Hpc application execution on grids. In *Dagstuhl Seminar Future Generation Grid 2004*, CoreGRID series. Springer, 2005. To appear.

[10] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organization. *The Intl. Journal of High Performance Computing Applications*, 15(3):200–222, Fall 2001.

[11] I. Foster and C. Kesselmann, eds. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, Dec. 2003.

[12] 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.

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

[14] C. A. R. Hoare. Communicating Sequential Processes. *Communications of ACM*, 21(8):666–677, Aug. 1978.

[15] J. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kauffman, 1993.

[16] S. Vadhiyar and J. Dongarra. Self adaptability in grid computing. *International Journal Computation and Currency: Practice and Experience*, 2005. To appear.

[17] R. V. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann,

and H. E. Bal. Ibis: a flexible and efficient Java-based grid programming environment. *Concurrency & Computation: Practice & Experience*, 2005.

[18] M. Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12):1709–1732, Dec. 2002.

Appendix A: ASSIST C4.5 code schema

```
typedef struct { ... } task_t;

/* --------------------- graph of modules definition -- */
generic main() {
  stream task_t s1; stream task_t s2;
  stream task_t s3; stream task_t s4;
  start  (                     output_stream  s1     );
  DC_c45 ( input_stream  s1, s2 output_stream  s3, s4 );
  CS_c45 ( input_stream  s3     output_stream  s2     );
  end    ( input_stream s4                            );
}

/* ----------------------------- sequential modules -- */
start(  output_stream task_t start_out )
{ start_proc (  out start_out ); }

end( input_stream task_t end_in )
{ end_proc (in end_in );}

/* ------------------------------- parallel modules -- */
parmod DC_c45(input_stream  task_t stream_start,task_t stream_rec
              output_stream task_t stream_task,task_t stream_result){
  topology one  vp;  /* behave as sequential process */
  input_section {
    guard_start: on ,,stream_start {
      distribution stream_start  broadcast  to vp;}
    guard_recursion: on ,,stream_rec {
      distribution stream_rec  broadcast  to vp;}
  }
  virtual_processes {
    guard_start_elab(in guard_start out stream_task) {
      VP {fc45( in  stream_start output_stream  stream_task );}
    }
    guard_recursion_elab(in guard_recursion out stream_task,stream_result){
      VP {fc45( in stream_rec output_stream stream_task,stream_result);}
    }
  }
  output_section {
    collects stream_task from ANY vp;
    collects stream_result from ANY vp;}
}

parmod CS_c45(input_stream  task_t stream_task
              output_stream task_t stream_task_out ) {
  topology none  vp;  /* behave as farm */
  input_section {
    guard_task: on ,,stream_task {
      distribution stream_task  on_demand  to vp;}
  }
  virtual_processes {
    guard_task_elab(in guard_task out stream_task_out) {
      VP {proc_compute(in stream_task output_stream stream_task_out );}
    }
  }
  output_section {collects stream_task_out from ANY vp;}
}

/* -------- sequential functions (procs) declaration -- */
proc start_proc ( out task_t start_out )  $c++{  ... }c++$
proc fc45 (in task_t task_in output_stream task_t task_out )  $c++{  ... }c++$
proc proc_compute (in task_t task_in output_stream task_t task_out) $c++{ }c++$
proc end_proc (in task_t end_in )  $c++{  ... }c++$
```