Automatic mapping of ASSIST applications using process algebra

Marco Aldinucci
Dept. of Computer Science, University of Pisa
Largo B. Pontecorvo 3, Pisa I-56127, Italy

and

Anne Benoit
LIP, Ecole Normale Superieure de Lyon (ENS)
46 allée d'Italie, 69364 Lyon Cedex 07, France

Received (received date)
Revised (revised date)
Communicated by (Name of Editor)

ABSTRACT
Grid technologies aim to harness the computational capabilities of widely distributed collections of computers. Due to the heterogeneous and dynamic nature of the set of grid resources, 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.

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

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) [13]. 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 programming 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 according to a suitable mapping onto grid platforms. These may be highly heterogeneous in architecture and performance and unevenly connected, thus exhibiting different connectivity properties among all 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 [19], eSkel [9], GrADS [17], ProActive [6], Ibis [18]. 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 developing tools and their run-time systems deal with the interaction with the grid, through collective protocols and services [12].

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 environment 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 [5]. 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.

This problem can be addressed 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 [15], 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 support. Section 3 introduces the Performance Evaluation Process Algebra PEPA, which can be used to model ASSIST applications. These performance models help to optimise the mapping of the application. We present our approach in Section 4, and give an overview of future working directions. Finally, concluding remarks are given in Section 5.

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 [19,3]. ASSIST applications should be compiled in binary packages that 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. 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, Java).

A *parmod* may behave in a data-parallel (e.g. SPMD/apply-to-all) or task-parallel (e.g. farm) way and it may exploit a distributed shared state that survives the 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 4–5 four streams with type `task_t` are declared. Lines 6–9 define endpoints of streams. Overall,

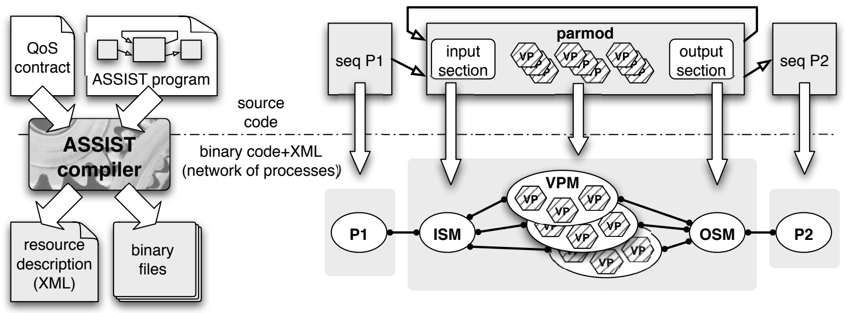

Fig. 1. An ASSIST application and a QoS contract are compiled in a set of executable codes and its meta-data [3]. This information is used to set up a processes network at launch time.

lines 3–10 define the application graph of modules. In lines 12–16 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 18–52 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 VPs as farm (topology `none`, as in line 41), 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 [16]. The `virtual_processes` declarations enable the programmer to realise a parametric VP 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 [19,3,2].

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. All processes communicate via ASSIST support channels, which 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 [3].

2.3. Towards fully grid-aware applications

ASSIST applications can already cope with platform heterogeneity, either in space (various architectures) or in time (varying load) [5,2]. These are definite features of a grid, however they are not the only ones. Grids are usually organised in sites on which processing elements are organised in networks with private addresses 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). Advance 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 our 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 we can analyse how to map an ASSIST application, assuming that we can exploit middleware tools to deploy and launch applications [11].

3. Introduction to performance evaluation and PEPA

In this section, we briefly introduce the Performance Evaluation Process Algebra PEPA [15], with which we can model an ASSIST 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. We can for instance define constants ($\stackrel{\mathrm{def}}{=}$), express the sequential behavior of a given component (.), a choice between different behaviors (+), and the direct interaction between components ($\underset{L}{\bowtie}$, $||$). 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. 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 [15]. 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 [15].

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 [14] 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

PEPA can easily be used to model an ASSIST application since such applications are based on stream communications, and the graph structure deduced from these streams can be modelled with PEPA. Given the probabilistic information about the performance of each of the ASSIST modules and streams, we then aim to find information about the global behavior of the application, which is expressed by the steady-state of the system. The model thus allows us to predict the run-time behavior of the application in the long time run, taking into account information obtained from a static analysis of the program. This behavior is not known in advance, it is a result of the PEPA model.

4.1. The ASSIST application

As we have seen in Section 2, an ASSIST application consists of a series of modules and streams connecting the modules. The structure of the application is represented by a graph, where the modules are the nodes and the streams the arcs. We illustrate in this paper our modeling process on an example of a graph, but the process can be easily generalised to any ASSIST applications since the

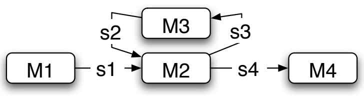

Figure 2: Graph representation of our example application.

information about the graph can be extracted directly from ASSIST source code, and the model can be generated automatically from the graph. A model of a data mining classification algorithm has been presented in [1]. For the purpose of our methodology and in order to generalise our approach, we concentrate here only on the graph of an application. The graph of the application that we consider in this paper is similar to the one of [1], consisting of four modules. Figure 2 represents the graph of this application.

We choose this graph as an application example, since this is a very common workflow pattern. In such a schema,

- one module (M1) is generating input, for instance reading from a file or accessing a database;
- two modules (M2, M3) are interacting in a client-server way; they can interact one or several times for each input, in order to produce a result;
- the result is sent to a last module (M4) which is in charge of the output.

4.2. The PEPA model

Each ASSIST module is represented as a PEPA component, and the different components are synchronised through the streams of data to model the overall application. The PEPA components of the modules are shown in Table 1. The modules are working in a sequential way: the module `MX` (`X`$=$ 1..4) is initially in the state `MX1`, waiting for data on its input streams. Then, in the state `MX2`, it processes the piece of data and evolves to its third state `MX3`. Finally, the module sends the output data on its output streams and goes back into its first state. The system evolves from one state to another when an activity occurs. The activity `sX` ($X = 1..4$) represents the transfer of data through the stream `X`, with the associated rate λ_X. The rate reflects the complexity of the communication. The activity `pX` ($X = 1..4$) represents the processing of a data by module `MX`, which is done at a rate μ_X. These rates are related to the theoretical complexity of the modules. A discussion on rates is done in Section 4.3.

The overall PEPA model is then obtained by a collaboration of the different modules in their initial states: $M11 \underset{s1}{\bowtie} M21 \underset{s2,s3}{\bowtie} M31 \underset{s4}{\bowtie} M41$.

4.3. Automatic generation of the model

The PEPA model is automatically generated from the ASSIST source code. This task is simplified thanks to some information provided by the user directly in the source code, and particularly the rates associated to the different activities of the PEPA model.

The rates are directly related to the theoretical complexity of the modules and of the communications. In particular, rates of the communications depend on: a) the speed of the links and b) data size and communications frequencies. A module may include a parallel computation, thus its rate depends on a) computing power of the platforms running the module and b) parallel computation complexity, its size, its parallel degree, and its speedup. Observe that aspect a) of both modules and communications rates strictly depends on mapping, while aspect b) is much more dependent on the application's logical structure and algorithms.

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. This information is defined directly in the ASSIST source code of the application by calling a `rate` function, in the body of the `main` procedure of the application (Appendix A, between lines 9 and 10). This function takes as a parameter the name of the modules and streams, and it should be called once for each module and each stream to fix the rates of the corresponding PEPA activities. 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)=(10,1000); rate(s2)=(10,1);    rate(s3)=(10,1); rate(s4)=(10,1000);
rate(M1)=(100,100); rate(M2)=(100,100); rate(M3)=(1,1);  rate(M4)=(100,100);
```

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. Details on the different types of guards can be found in [19,3].

Table 1: PEPA model for the example

$$\begin{array}{lcl}
M11 & \stackrel{\text{def}}{=} & M12 \\
M12 & \stackrel{\text{def}}{=} & (p1,\mu_1).M13 \\
M13 & \stackrel{\text{def}}{=} & (s1,\lambda_1).M11 \\
 & & \\
M21 & \stackrel{\text{def}}{=} & (s1,\top).M22 + (s2,\top).M22 \\
M22 & \stackrel{\text{def}}{=} & (p2,\mu_2).M23 \\
M23 & \stackrel{\text{def}}{=} & (s3,\lambda_3).M21 + (s4,\lambda_4).M21
\end{array}
\qquad
\begin{array}{lcl}
M31 & \stackrel{\text{def}}{=} & (s3,\top).M32 \\
M32 & \stackrel{\text{def}}{=} & (p3,\mu_3).M33 \\
M33 & \stackrel{\text{def}}{=} & (s2,\lambda_2).M31 \\
 & & \\
M41 & \stackrel{\text{def}}{=} & (s4,\top).M42 \\
M42 & \stackrel{\text{def}}{=} & (p4,\mu_4).M43 \\
M43 & \stackrel{\text{def}}{=} & M41
\end{array}$$

As an example, a 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 also be expressed, for instance the parmod can be instructed to 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, $M21 \stackrel{\text{def}}{=} (s1, \top).(s2, \top).M22 + (s2, \top).(s1, \top).M22$. Currently, we are not supporting 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. 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 application for a given set of rates is represented below:

```
mu1=100;mu2=100;mu3=1;mu4=100;
la1=10;la2=10;la3=10;la4=10;

M11=M12; M12=(p1,mu1).M13; M13=(s1,la1).M11;
M21=(s1,infty).M22 + (s2,infty).M22; M22=(p2,mu2).M23;
M23=(s3,la3).M21 + (s4,infty).M21;
M31=(s3,infty).M32; M32=(p3,mu3).M33; M33=(s2,la2).M31;
M41=(s4,la4).M42; M42=(p4,mu4).M43; M43=M41;

(M11 <s1> (M21 <s2,s3> M31)) <s4> M41
```

4.4. Performance results

Once the PEPA models have been generated, performance results can be obtained easily with the PEPA Workbench [14]. The performance results are the probability to be in either of the states of the system. We compute the probability to be waiting for a processing activity `pX`, or to wait for a transfer activity `sX`. Some additional information is generated in the PEPA source code (file `example.pepa`) to specify the performance results that we are interested in. This information is the following:

```
perf_M1= 100 * {M12 || ** || **  || **}; perf_M2= 100 * {** || M22 || ** || ** };
perf_M3= 100 * {**  || ** || M32 || **}; perf_M4= 100 * {** || **  || ** || M42};
perf_s1= 100 * {M13 || M21|| **  || **}; perf_s2= 100 * {** || M21 || M33|| ** };
perf_s3= 100 * {**  || M23|| M31 || **}; perf_s4= 100 * {** || M23 || ** || M41};
```

The expression in brackets describes the states of the PEPA model corresponding to a particular state of the system. For each module `MX` (`X` $= 1..4$), the result `perf_MX` 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 [14].

Experiment 1. For the purpose of our example, we choose the following rates, meaning that the module $M3$ is computationally more intensive than the other modules. In our case, $M3$ has an average duration of 1 sec. compared to 0.01 sec. for the others ($\mu_1 = 100; \mu_2 = 100; \mu_3 = 1; \mu_4 = 100$). The rates for the streams correspond to an average duration of 0.1 sec ($\lambda_1 = 10; \lambda_2 = 10; \lambda_3 = 10; \lambda_4 = 10$). The results for this example are shown in Table 2 (row *Case 1*).

These results confirm the fact that most of the time is spent in module M3, which is the most computationally demanding. Moreover, module M1 (respectively M4) spends most of its time waiting to send data on s1 (respectively waiting to receive data from s4). M2 is computing quickly, and this module is often receiving/sending from stream s2/s3 (little time spent waiting on these streams in comparison with streams s1/s4).

If we study the computational rate, we can thus decide to map M3 alone on a powerful computing site because it has the highest value between the different steady states probabilities of the modules. One should be careful to map the streams s1 and s4 onto sufficiently fast network links to increase the overall throughput of the network. A mapping that performs well can thus be deduced from this information, by adjusting the reasoning to the architecture of the available system.

Experiment 2. We can reproduce the same experiment but for a different application: one in which there are a lot of data to be transfered inside the loop. Here, for one input on $s1$, the module M2 makes several calls to the server M3 for computations. In this case, the rates of the streams are different, for instance $\lambda_1 = \lambda_4 = 1000$ and $\lambda_2 = \lambda_3 = 1$.

The results for this experiment are shown in Table 2 (row *Case 2*). In this table, we can see that M3 is quite idle, waiting to receive data 89.4% of the time (i.e. this is the time it is not processing). Moreover, we can see in the stream results that s2 and s3 are busier than the other streams. In this case a good solution might be to map M2 and M3 on to the same cluster, since M3 is no longer the computational bottleneck. We could thus have fast communication links for s2 and s3, which are demanding a lot of network resources.

Table 2: Performance results for the example.

	Modules				Streams			
	M1	M2	M3	M4	s1	s2	s3	s4
Case 1	4.2	5.1	67.0	4.2	47.0	6.7	6.7	47.0
Case 2	52.1	52.2	10.6	52.1	5.2	10.6	10.6	5.2

4.5. Analysis summary

As mentioned in Section 4.3, PEPA rates model both aspects strictly related to the mapping and to the application's logical structure (such as algorithms implemented in the modules, communication patterns and size). The predictive analysis conducted in this work provides performance results which are related only to the application's logical behavior. On the PEPA model this translates on the assumption that all sites includes platforms with the same computing power, and all links have an uniform speed. In other words, we assume to deal with a homogeneous grid to obtain the relative requirements of power among links and platforms. This information is used as a hint for the mapping on a heterogeneous grid.

It is of value to have a general idea of a good mapping solution for the application, and this reasoning can be easily refined with new models including the mapping peculiarities, as demonstrated in our previous work [1]. However, the modeling technique exposed in the present paper allows us to highlight individual resources (links and processors) requirements, that are used to label the application graph.

These labels represent the expected relative requirements of each module (stream) with respect to other modules (streams) during the application run. In the case of a module the described requirement can be interpreted as the aggregate power of the site on which it will be mapped. On the other hand, a stream requirement can be interpreted as the bandwidth of the network link on which it will be mapped. The relative requirements of parmods and streams may be used to implement mapping heuristics which assign more demanding parmods to more powerful sites, and more demanding streams to links exhibiting higher bandwidths. When a fully automatic application mapping is not required, modules and streams requirements can be used to drive a user-assisted mapping process.

Moreover, each parmod exhibits a structured parallelism pattern (a.k.a. skeleton). In many cases, it is thus possible to draw a reliable relationship between the site fabric level information (number and kind of processors, processors and network benchmarks) and the expected aggregate power of the site running a given parmod exhibiting a parallelism pattern [5,4,8]. This may enable the development of a mapping heuristic, which needs only information about sites fabric level information, and can automatically derive the performance of a given parmod on a given site.

The use of models taking into account both of the system architecture characteristics can then eventually validate this heuristic, and give expected results about the performance of the application for a specified mapping.

4.6. Future work

The approach described here considers the ASSIST modules as blocks and does not model the internal behavior 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 [7,8], 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).

We believe that this approach is particularly suitable for modeling applications that can be described by a graph, not just ASSIST applications (such as applications described in the forthcoming CoreGrid Grid Component Model [10]). In particular the technique described here helps to derive some information about the pressure (on modules and links) within a loop of the graph. Loops are quite common patterns; they can be used to describe simple interactions between modules (e.g. client-server RPC behavior) or mutual recursive dependency between modules. These two cases lead to very different behaviors in term of pressure or resources within the loop; in the former case this pressure is variable over time.

The mapping decision is inherently a static process, and especially for loops in the graph, it is important to make decisions on the expected common case. This is modeled by the PEPA steady state probabilities, that indeed try to give some static information on dynamic processes.

5. Conclusions

In this paper we have presented a method 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. 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.

Acknowledgements. The authors are supported by the **Enhance** project of the university of Edinburgh, funded by the EPSRC (grant GR/S21717/01); the Italian MIUR FIRB **Grid.it** project (RBNE01KNFP); the FP6 Network of Excellence **CoreGRID**, funded by the European Commission (contract IST-2002-004265).

References

[1] M. Aldinucci and A. Benoit. Automatic mapping of ASSIST applications using process algebra. Technical Report TR-0016, Institute on Programming Model, CoreGRID -

Network of Excellence, Oct. 2005.

[2] M. Aldinucci, M. Coppola, S. Campa, M. Danelutto, M. Vanneschi, and C. Zoccolo. Structured implementation of component based grid programming environments. In *Future Generation Grids*, CoreGRID series, pp. 217–239. Springer Verlag, Nov. 2005.

[3] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, and C. Zoccolo. ASSIST as a research framework for high-performance grid programming environments. In J. C. Cunha and O. F. Rana, editors, *Grid Computing: Software environments and Tools*, chapter 10, pp. 230–256. Springer Verlag, Jan. 2006.

[4] M. Aldinucci, M. Danelutto, J. Dünnweber, and S. Gorlatch. Optimization techniques for skeletons on grid. In L. Grandinetti, editor, *Grid Computing and New Frontiers of High Performance Processing*, vol. 14 of *Advances in Parallel Computing*, chapter 2, pp. 255–273. Elsevier, Oct. 2005.

[5] M. Aldinucci, M. Danelutto, and M. Vanneschi. Autonomic QoS in ASSIST grid-aware components. In *Proc. of Intl. Euromicro PDP 2006: Parallel Distributed and network-based Processing*, pp. 221–230, Montbéliard, France, Feb. 2006. IEEE.

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

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

[8] 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.

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

[10] CoreGRID NoE deliverable series, Institute on Programming Model. *Deliverable D.PM.02 – Proposals for a Grid Component Model*, Nov. 2005.

[11] M. Danelutto, M. Vanneschi, C. Zoccolo, N. Tonellotto, S. Orlando, R. Baraglia, T. Fagni, D. Laforenza, and A. Paccosi. HPC application execution on grids. *Future Generation Grids*, CoreGRID series, pp. 263–282. Springer Verlag, Nov. 2005.

[12] 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.

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

[14] 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*, vol. 794 of *LNCS*, pp. 353–368, Vienna, May 1994. Springer Verlag.

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

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

[17] S. Vadhiyar and J. Dongarra. Self adaptability in grid computing. *Concurrency & Computation: Practice & Experience*, 17(2–4):235–257, 2005.

[18] 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*, 17(7-8):1079–1107, 2005.

[19] 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 code schema example

```
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;
  M1 (                       output_stream  s1     );
  M2 ( input_stream  s1, s2 output_stream  s3, s4 );
  M3 ( input_stream  s3     output_stream  s2     );
  M4 ( input_stream  s4                           );
}
/* ----------------------------------- sequential modules ----------- */
M1( output_stream task_t start_out )
{ proc_M1 (out start_out );}

M4( input_stream task_t end_in )
{ proc_M4 (in end_in );}
/* ----------------------------------- parallel modules ------------- */
parmod M2(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 {proc_M2( in stream_start out stream_task);}
    }
    guard_recursion_elab(in guard_recursion out stream_task,stream_result){
      VP {proc_M2( in stream_rec out stream_task,stream_result);}
    }
  }
  output_section {
    collects stream_task   from ANY vp;
    collects stream_result from ANY vp;}
}
parmod M3(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_M3(in stream_task out stream_task_out );}
    }
  }
  output_section {collects stream_task_out from ANY vp;}
}
/* -------- sequential functions (procs) declaration -- */
proc proc_M1 (                out task_t start_out) $c++{ ... }c++$
proc proc_M2 (in task_t task_in out task_t task_out ) $c++{ ... }c++$
proc proc_M3 (in task_t task_in out task_t task_out ) $c++{ ... }c++$
proc proc_M4 (in task_t end_in )                      $c++{ ... }c++$
```