Scalable Computing: Practice and Experience
Volume 11, Number 3, pp. 263–275. http://www.scpe.org

ISSN 1895-1767
© 2010 SCPE

MODELING STREAM COMMUNICATIONS IN COMPONENT-BASED APPLICATIONS *

M. DANELUTTO†, D. LAFORENZA‡, N. TONELLOTTO‡, M. VANNESCHI†, AND C. ZOCCOLO†

Abstract. Component technology is a promising approach to develop Grid applications, allowing to design very complex applications by hierarchical composition of basic components. Nevertheless, component applications on Grids have complex deployment models. Performance-sensitive decisions should be taken by automatic tools, matching developer knowledge about component performance with QoS requirements on the applications, in order to find deployment plans that satisfy a Service Level Agreement (SLA).

This paper presents a steady-state performance model for component-based applications with stream communication semantics. The model strictly adheres to the hierarchical nature of component-based applications, and is of practical use in launch-time decisions.

Key words: grid computing; heterogeneous environments; stream computations; performance model; mapping.

1. Introduction. Grid computing is an emerging technology that enables the aggregation of heterogeneous, distributed resources to solve computational problems of ever increasing size and complexity. The applications that best perform on Grid platforms are the ones requiring large computational power, or the treatment of large data sets, i. e. a subclass of High-Performance Applications [17].

Such applications (e.g. data-mining [12], query processing [3], image processing and visualization [2] and multimedia streaming [38]) can be conveniently expressed using a formalism based on two fundamental notions: streams of data flowing between components, and components (either sequential or parallel) processing them. Several programming languages are built on these concepts. Skeleton-based languages (e.g. SkIE [4] and SBASCO [14]) and skeleton libraries (e.g. eSkel [11] and Kuchen's C++ skeleton library [21]) exploit the notion of streams for task-parallel skeletons (e.g. pipe and farm). More general languages like ASSIST [33] and Datacutter [15] introduce modules and streams as primitive concepts to structure parallel applications.

Grid programming frameworks (e.g. GrADS [9], ASSIST [13]) are in charge of the complete automation of application execution management, efficiently exploiting Grid resources. Moreover, they should be able to execute the application with user-required QoS, adapting the execution to the dynamic changes of Grid resources.

The traditional component mapping strategy, in which components are *statically* deployed in a distributed environment by their developers, does not fit well in such scenario. A broader deployment model is required, featuring

(i) manual mapping, in which the components are already paired with their resources (on which they are deployed),

(ii) resources discovery and selection at launch time, to guarantee the initial desired performance,

(iii) adaptive components management, that at run-time adjust the set of computing resources exploited [31, 1], in order to adapt to different performance requirements (on-demand computing) or to changing resources availability.

According to this model, the deployment framework must automatically manage the operations needed to enforce the application desired QoS. This can be obtained with the specification of a *performance contract* [34].

Our approach intends to automatise the tasks needed to start the execution of HPC applications. Our final goal is to allow an as large as possible user community to gain full benefits from the Grid, and at the same time to give the maximum generality, applicability and easy of use.

The main contributions of this paper are as follows:

(i) We propose an analytical model of the dynamic behavior of sequential/parallel components, hierarchical components and component applications, communicating through typed streams of data. It is suited to be used in simulation environments, to synthetically generate components and applications to test mapping/scheduling solutions in a repeatable and controlled setting. Eventually, the proposed dynamic model can be exploited in the implementation of dynamic reconfiguration policies [1].

*This work has been supported by: the Italian MIUR FIRB Grid.it project, No. RBNE01KNFP, on High-performance Grid platforms and tools, and the European CoreGRID NoE (European Research Network on Foundations, Software Infrastructures and Applications for Large Scale, Distributed, GRID and Peer-to-Peer Technologies, contract no. IST-2002-004265).

†Department of Computer Science, University of Pisa, Pisa, Italy

‡Information Science and Technologies Institute, National Research Council, Pisa, Italy

(ii) Starting from the dynamic model we identify the set of variables that can be used to describe the performance behavior of an application, and we derive the set of relations among them which hold at steady-state (performance model). In this way we abstract from particular runtime platforms and we capture all possible steady-state behaviors of an application. Moreover, their formulation by means of linear algebra allows us to hierarchically compose the performance models of several components to derive the steady-state model of new components or applications.

(iii) We introduce a definition of *performance model* for stream applications, which is exploited in launch-time mapping and runtime reconfiguration decisions.

After a survey of related work (Sect. 2), this paper presents a dynamic model of stream-based computations (Sect. 3), and in Sect. 4 such model is exploited to derive a steady-state performance model for stream-based applications. In Sect. 5, such model is applied to a case study, to predict the program behavior at run-time, and to devise a correct initial mapping for specified QoS levels. Section 6 concludes the paper, discussing the presented approach and future work.

2. Related Work. Performance specification of components and their interactions is a basic problem that must be solved to enable software engineers to assemble efficient applications [27]. Moreover, performance modeling is one of the key aspects that needs to be addressed to face scheduling/mapping problems in heterogeneous platforms. It arises in automatic component placement and reconfiguration. Several recent works focus on performance modeling techniques to analyze the behavior of component-based parallel applications on distributed, heterogenous, dynamic platforms.

Analytic performance models in software engineering make extensive use of UML formalism to describe software component behavioral models [35] and to derive models based on Queuing Networks [19] or Layered Queueing Networks [36] to be exploited in design phase of the lifecyle of software. The same holds for Stochastic Petri Nets [20] and Stochastic Process Algebras [18]. Such models typically translate a parallel application into an analytic representation of its execution behavior and the target runtime system (according to the Software Performance Engineering methodology [28]). A detailed survey of such models is in [5]. Such translation is usually not straightforward. It may require approximations to obtain mathematical models [29] for which a closed-form solution is known. Stochastic models usually require the solution of the underlying Markov chain which can easily lead to numerical problems due to the space state explosion [5]. More complex models can be solved by means of simulation, at the cost of a larger computation time.

Symbolic performance modeling [32] is a methodology that enables a rapid development of low complexity and parametric performance models. Symbolic performance models can be derived from simulation models, trading off result accuracy for model evaluation cost. In [32] a symbolic performance model for the Pamela modeling language is introduced. It derives lower bounds for steady-state performances of applications starting from a model of the program and of the shared resources, combining deterministic Direct Acyclic Graphs (DAGs) modeling with mutual exclusion. One of the strengths of the Pamela approach is that it is fast and easy to transform a regularly structured application into a performance model. The main limitation of such approach is that it computes lower bounds of the performance of a program. Symbolic performance models share several properties with the model we propose: both can be extracted from the structure of programs, are parametric, and can be efficiently evaluated. The main difference is that the presented model does not compute a lower bound, but the asymptotic steady-state performance of an application, that is in general a better approximation of the real performance.

The asymptotic steady-state analysis has been pioneered by Bertsimas and Gamarnik [10]. This approach has been recently applied to mapping and scheduling problems of parallel applications on heterogeneous platforms [23, 7, 6], in which the analysis is applied to particular classes of parallel applications (divisible load [23], master/slave [6], pipelined and scatter operations [7]), in the hypothesis that the set of resources is known in advance. The existing steady-state approaches apply only to a restricted class of structured parallel applications, assuming to know the runtime environment in such a way to derive optimal scheduling of the application components. In a dynamic environment like a Grid an optimal initial placement of the components may become useless very soon, because the conditions of the execution platform may vary dynamically. The presented steady-state analysis can be applied to a broader class of structured parallel applications and tries to solve a different problem, i. e. to build a concrete model of components/applications to be exploited in their mapping on previously-unknown target platforms.

Structural performance models [25] are the first effort to develop compositional performance models for component applications. Most scientific and Grid component models rely on the concept of algorithmic skeleton. Skeletons are common, reusable and efficient structured parallelism exploitation patterns. One advantage of the skeletal approach is that parametric cost models can be devised for the evaluation of runtime performance of skeleton compositions. In [14, 8] different cost models are associated to each skeleton of an application to enhance its runtime performance through parallelism/replication degree adjustments and initial mapping selection, respectively. The authors of [14] propose parametric cost models for PIPE, FARM and MULTIBLOCK skeletons, that can be arbitrarily composed and nested. In [8], analytic cost models for applications composed by PIPEs and DEALs are derived within a stochastic process algebra formulation. Structural performance models are extended by the presented model by proposing a methodology well-suited for generic composition of skeletons, and by taking into account the synchronization problems introduced by using streamed communications.

Trace-based performance models [34, 26] are currently exploited in parallel/Grid environments to model the performance of sets of kernel applications. Recording and analyzing execution traces on reference architectures of such application it is possible, with a certain degree of precision, to forecast the performance of the same or similar applications on different resources. Trace information is exploited in the presented model, but in different way with respect to the existing approaches. Instead of profiling a whole application on a set of representative resources, the application model is kept independent from resources. When the application will be mapped on actual resources, historical information will be used to model the runtime behavior of single components, and then such information will be coupled with the component interactions information to obtain a prediction of the performance of the whole application.

The problem of deriving a performance model for components has been addressed also in the context of component frameworks such as EJB [37], COM+/.NET [16] and CCA [24]. Such works apply analytical performance model (LQN) or trace-based performance model to derive a model for components. In [30], trace-based models are exploited to select the most suitable components, when multiple choices are available, to build an optimal application, from the point of view of performance.

3. Dynamic Behavior. An application can be structured as a hypergraph whose nodes represent primitive components and whose (hyper)edges represent communications or synchronizations between components. Nodes interact with input (server) interfaces and output (client) interfaces. Edges are directed and can connect two or more nodes through their interfaces. Two nodes may be linked by more than a single edge.

3.1. Communications. Communications between components are implemented through input/output interfaces bindings. In this work data-flow stream communications are studied. Every component receives data through one or more input interfaces, performs some computations, and generates new data to be sent through one or more output interfaces.

In this context, a *stream* represents a typed, unidirectional communication channel between a non-empty, finite set of components (producers) and a non-empty, finite set of components (consumers). The atomic piece of information transferred through a stream is called *item*. A producer is connected to a stream through an output interface, while a consumer is connected to a stream through an input interface. Every node can be producer or consumer of several streams, and it is possible to specify cyclic structures (i. e. the communication structure is not restricted to be a DAG).

Components can be connected by streams according to three different patterns:

(i) **unicast**: one-to-one connection. Every item sent on the output stream interface is received in order by the input stream interface.

(ii) **merge**: many-to-one connection. Every item sent on the output stream interfaces is received by the input stream interface. The temporal ordering of the items coming from each input interface is preserved, but the interleaving between the different sources is non-deterministic.

(iii) **broadcast**: one-to-many connection. Every item sent on the output stream interface is received in order by the input stream interfaces. The receptions happening on different input interfaces are not synchronized.

3.2. Computations. Components implement sequential as well as parallel computations. A sequential component executes a single function in a single active thread, processing items as they are received. For a parallel component, two scenarios are possible:

(i) **data parallel**: a single function is executed in parallel on different portions of the same data;

(ii) **task parallel**: several functions (or activations of the same function) are executed in parallel on independent data.

A primitive component, either sequential or parallel, at runtime repeatedly receives items from its input streams, performs some computations and delivers result items to its output streams.

A component can have several input streams. The set of input streams is partitioned between the computations associated with the components. Each input stream is associated to only one computation; nevertheless, spontaneous computations may exist, that do not need input items to activate, but follow own activation policies (e.g. periodically).

A computation can be activated if the following conditions hold:

(i) the component can execute a new function (this means that it is idle, or it is parallel and threads are available to execute it),

(ii) the associated input items have been received, or no item is necessary.

A sequential component can activate a new function only when it is idle. A parallel component can have at most one active data-parallel computation at any given time (composed by a fixed number of threads), or several task-parallel computations running in parallel (up to the maximum number of threads in the component).

A component can have several output streams. One or more computations of the component can dispatch data on each output stream.

3.3. Node Behavior. In order to describe the behavior of a computation at runtime, consider Fig. 3.1.

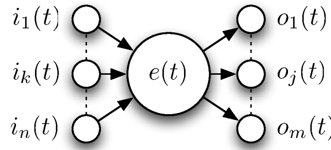

Fig. 3.1. *Sequential component at runtime*

Without loss of generality, a sequential component is considered; the displayed quantities represent:

(i) $i_k(t)$: total number of received items at time t from the kth input interface;

(ii) $e(t)$: total number of computations carried out at time t;

(iii) $o_j(t)$: total number of sent items at time t through the jth output interface.

Continuous quantities are used to model partial evolution, e.g. $e(t) = 3.5$ means that the node reached the half way point in the fourth computation.

The activation of a computation can happen only when the number of items completely received on each associated stream is greater than the number of partially computed items:

$$\forall k = 1, \dots, n \qquad \lfloor i_k(t) \rfloor - e(t) > 0 \tag{3.1}$$

The node implementation will exploit finite buffers to store received items for each input interface, therefore for each input interface and associated computation the following must hold:

$$\forall k = 1, \dots, n \qquad i_k(t) - \lfloor e(t) \rfloor \leq \tau_{1k} \tag{3.2}$$

where τ_{1k} represents the maximum number of elements that can be received on the kth input interface before the stream blocks. Then the maximum admissible value for $i_k(t)$ at time t is:

$$i_k^{max}(t) = \tau_{1k} + \lfloor e(t) \rfloor \tag{3.3}$$

Assuming that no sensible delays are present between the end of computations and the beginning of the transmission of the produced items, the total number of transmitted items is related to the progress of the computations of the node. In the general case of a node with s functions, the following equation holds for each output interface:

$$\forall j = 1, \dots, m \qquad o_j(t) = f_j\big(e_1(t), \dots, e_s(t)\big) \tag{3.4}$$

where $e_i(t)$ represents the number of activations carried out at time t for the $i - th$ function. The *transfer function* f_j relates the number of data outputs $o_j(t)$ to the number of performed computations $e_1(t), \dots, e_s(t)$.

3.4. Edge Behavior. In order to describe the behavior of a data transmission on a stream, consider a unicast stream. The involved variables are $o(t)$, total number of items sent at time t from source interface, and $i(t)$, total number of items received at time t by the destination interface. A new transmission begins only after a full item is produced:

$$i(t) \leq \lfloor o(t) \rfloor \tag{3.5}$$

The edge implementation will exploit finite communication buffers and the network layer transfers chunks of data. Let q^{-1} be the minimum fraction of item transferred atomically. Then

$$o(t) - \frac{\lfloor q \cdot i(t) \rfloor}{q} \leq \tau_2 \tag{3.6}$$

where τ_2 represents the maximum number of items that can be buffered. Therefore the maximum admissible value for $o(t)$ at time t is:

$$o^{max}(t) = \tau_2 + \frac{\lfloor q \cdot i(t) \rfloor}{q} \tag{3.7}$$

Whenever an edge buffer is full, a producer will block as soon as it tries and sends a new item. From (3.4) we obtain:

$$o^{max}(t) - f\big(e_1(t), \dots, e_m(t)\big) \leq 0 \tag{3.8}$$

For *merge* streams with k source interfaces and *broadcast* streams with k destination interfaces, the general constraints (Eqs. (3.5) and (3.6) for the unicast stream) become:

$$\text{merge: } \begin{cases} i(t) \leq \sum_k o_k(t) \\ \sum_k o_k(t) - i(t) \leq \tau_{2k} \end{cases} \tag{3.9}$$

$$\text{broadcast: } \begin{cases} \forall k \quad i_k(t) \leq o(t) \\ \forall k \quad o(t) - i_k(t) \leq \tau_{2k} \end{cases} \tag{3.10}$$

For simplicity, in the previous equations the network quantization constant q has been suppressed.

3.5. Runtime Behavior. At runtime, a component can be seen as a dynamic system. The system state at time t is described by a set of state variables: $i_{1,\dots,n_i}(t)$, $e_{1,\dots,n_e}(t)$, $o_{1,\dots,n_o}(t)$. Thus, the state space $\mathbb{P}$ is a $n = n_i + n_e + n_o$ dimension Euclidean space. The dynamic behavior of a component can be modeled by a trajectory $p(t)$ in such state space.

The runtime behavior of a component is fully specified when it is coupled with hosting resources. A computing resource is modeled by $w(t)$, the available computing power at time t (measured in MFlop/s) and a communication link is modeled by $b(t)$, the instantaneous bandwidth at time t (measured in MByte/s). Moreover, a characterization of the items is required. It is assumed that an item processed by a component requires l units of computing work to be processed (measured in MFlop) and s units of communication work to be transmitted (measured in bytes).

Introducing the *step function* $u(x)$, the number of performed (partial) computations per time unit is:

$$\begin{aligned} \frac{de}{dt} = u\Big(\min\big(\lfloor i_1(t) \rfloor, \dots, \lfloor i_n(t) \rfloor\big) - e(t)\Big) \cdot \\ \cdot\, u\Big(o^{max}(t) - f\big(e_1(t), \dots, e_m(t)\big)\Big) \cdot \frac{w(t)}{L} \end{aligned} \tag{3.11}$$

while the equations governing the number of packets flowing in the unicast, merge and broadcast streams per time unit are, respectively:

$$\frac{di}{dt} = u\Big(\lfloor o(t) \rfloor - i(t)\Big) \cdot u\Big(i^{max}(t) - i(t)\Big) \cdot \frac{b(t)}{s} \tag{3.12a}$$

$$\frac{di}{dt} = u\Big(\sum_k \lfloor o_k(t) \rfloor - i(t)\Big) \cdot u\Big(i^{max}(t) - i(t)\Big) \cdot \frac{b(t)}{s} \tag{3.12b}$$

$$\frac{di_k}{dt} = u\Big(\lfloor o(t) \rfloor - i_k(t)\Big) \cdot u\Big(i^{max}(t) - i_k(t)\Big) \cdot \frac{b(t)}{s} \tag{3.12c}$$

Note that an important assumption has been made. The work required to perform a computation is supposed to be *independent* from the values of the incoming items; their values are used just to perform computations. This is a common assumption in parallel data-flow programming, but there are applications (e.g. query processing and data mining) that do not respect this assumption.

The dynamic equations provided by the model can be written in the general form:

$$\dot{p}(t) = U(p(t))\,\alpha(t) \tag{3.13}$$

We denote with $U : \mathbb{P} \to \mathbf{M}_{n,n}$ the function that, for every point in the state space, provides the control part of the differential equations (the ones involving the step functions), and with $\alpha(t)$ the resources part (involving $w(t)$ and $b(t)$).

We observe that the control matrix is piece-wise constant over non-infinitesimal time intervals: it descends from quantization in the general equations for the nodes (3.11), and in the equations for the streams (3.12). Then, the Cauchy problem can be solved constructively. Starting with $t_0 = 0, p_0(t_0) = 0, U_0 = U(0)$, we inductively define

$$\begin{aligned} p_i(t) &= \int_{t_i}^{t} U_i \alpha(\tau) d\tau \\ t_{i+1} &= \sup\{t > t_i | U(p_i(t)) = U_i\} \\ U_{i+1} &= \lim_{t \to t_i^+} U(p_i(t)) \end{aligned}$$

In this way, $p(t)$ is defined as the concatenation of the pieces $p_i|_{[t_i,t_{i+1})}$: it is a continuous function ($p_i(t_i) = p_{i+1}(t_i)$) and piece-wise differentiable.

4. STEADY STATE BEHAVIOR. The steady-state behavior of the system can be analysed by studying mean values $\bar{p}$ for the rate of change of the state variables:

$$\bar{p} = \mathbb{E}[\dot{p}|_{[t_0,\infty)}] = \int_{t_0}^{\infty} \dot{p}(t) dt = \lim_{t \to \infty} \frac{p(t) - p(t_0)}{t - t_0} \tag{4.1}$$

The choice of t_0 is arbitrary, in fact the weight of the transient phase fades away considering infinite executions. However, to ease the reasoning about these quantities, we can interpret t_0 as the end of the transient phase, e.g. when the last stage consumes the first data item in a pipeline.

The essential aspect to point out is that for the steady-state model the focus is on relations among the steady-state variables, rather than in their values. In this way it is possible to abstract from particular target platforms, and capture the class of all possible steady-state behaviors of an application.

The steady-state behavior of a node can be modelled associating to each computation $e_k(t)$ its activation rate

$$\bar{e}_k = \lim_{t \to \infty} \frac{e_k(t) - e_k(t_0)}{t - t_0} \tag{4.2}$$

Spontaneous computations are free variables in the steady-state model. Computations that are activated by data reception, instead, are subject to the following condition.

Proposition 4.1. *The steady-state execution rate of a computation is bound to be equal to the input rates on the input interfaces that activate the computation.*

Proof. Let $k \in A_i$, we will prove that $\bar{e}_i - \bar{\imath}_k = 0$

$$\begin{aligned} \bar{e}_i - \bar{\imath}_k &= \lim_{t \to \infty} \frac{e_i(t) - e_i(t_0)}{t - t_0} - \lim_{t \to \infty} \frac{i_k(t) - i_k(t_0)}{t - t_0} \\ &= \lim_{t \to \infty} \frac{e_i(t) - e_i(t_0) - i_k(t) + i_k(t_0)}{t - t_0} \\ &= \lim_{t \to \infty} \frac{e_i(t) - i_k(t)}{t - t_0} - \frac{e_i(t_0) - i_k(t_0)}{t - t_0} \end{aligned}$$

The numerator of the first addend is limited by constants: (3.1) gives

$$e_i(t) - i_k(t) \leq 0$$

and (3.2) (noting that $e(t) \geq \lfloor e(t) \rfloor$) gives

$$e_i(t) - i_k(t) \geq -\tau_{1k}$$

while the numerator of the second addend is constant, so the limit tends to zero when the denominator tends to infinity. □

The data transmission rate $\bar{o}_k$ of an output stream will depend on the activation rates of one or more computations of the node. In the previous section, the number of data outputs has been related to the number of performed computations by means of a *transfer function* f_k (Eqn. (3.4)).

PROPOSITION 4.2. *If the transfer function is (asymptotically) linear*

$$o_k = f_k(e_1, \ldots, e_m) = \alpha_k^1 e_1 + \ldots \alpha_k^m e_m + c_k(e_1, \ldots, e_m)$$

with

$$\lim_{\|e\| \to \infty} \frac{\|c_k(e)\|}{\|e\|} = 0$$

then a steady-state is eventually reached, in which the output rate is a linear combination of the computation rates:

$$\bar{o}_k = \sum_{i=1}^{m} \alpha_{ki} \bar{e}_i \tag{4.3}$$

Proof.

$$\bar{o}_k = \lim_{t \to \infty} \frac{f_k(e(t)) - f_k(e(t_0))}{t - t_0} = \lim_{t \to \infty} \frac{\alpha_k \cdot (e(t) - e(t_0)) + c(e(t)) - c(e(t_0))}{t - t_0} =$$

$$\alpha_k \cdot \lim_{t \to \infty} \frac{e(t) - e(t_0)}{t - t_0} + \lim_{t \to \infty} \frac{c(e(t)) - c(e(t_0))}{t - t_0} = \alpha_k \cdot \bar{e} + 0 = \sum_{i=1}^{m} \alpha_k^m \bar{e}_i$$

□

The steady-state behavior of streams can be modelled by associating to each endpoint its data transmission rate. Balance equations relating input and output endpoints are derived.

PROPOSITION 4.3. *The steady-state transmission rate at the endpoints of a stream are characterised by the following balance equations:*

$$\textit{unicast: } \bar{o}_A = \bar{\imath}_B \tag{4.4a}$$

$$\textit{merge: } \bar{o}_A + \bar{o}_B = \bar{\imath}_C \tag{4.4b}$$

$$\textit{broadcast: } \bar{o}_A = \bar{\imath}_B = \bar{\imath}_C \tag{4.4c}$$

These equations are easily extended in the case of more endpoints.

Proof. The proof is similar to the one of Prop. 4.1, exploiting:

(i) (3.5) and (3.6) for unicast,

(ii) (3.9) for merge,

(iii) (3.10) for broadcast.

□

The *execution rate* for each computation, and the *data transfer rate* for each input/output interface completely specify the application state from the point of view of its performance, therefore we will call them the **performance features** of our application.

Proposition 4.2 allows us to express output rates as linear combinations of execution rates, provided that we know the related coefficients. These coefficients must be provided by developers of programs/components

by means of some **performance annotations**, in order to build a performance model. Proposition 4.1 allows us to eliminate execution rates associated to data-dependent computations. Proposition 4.3 allows us to relate output rates to input rates of linked modules.

The **performance model** is therefore defined as an homogeneous system of simultaneous linear equations, that describe the relations that hold in the steady-state among the **performance features**. The set of solutions of the system is a vector subspace of $\mathbb{R}^n$ (where n is the total number of variables, either input rates, output rates or execution rates); we call the dimension of the solution space the number of **degrees of freedom** of the application. If this dimension is 1, then the system is completely determined as soon as a single value for any variable is imposed. The degenerate case of a space with dimension 0 implies that the only solution to the system is the null vector (i. e. every variable must be zero): this means that the predicted steady-state is a deadlock state, in which no computation or communication can proceed. The number of degrees of freedom of the system will impact on how many constraints must be provided in order to derive the expected values for every variable.

Clearly, only positive values of the rates are meaningful, so we can conclude that every assignment of positive values for the vector $[\mathbf{i}\,\mathbf{e}\,\mathbf{o}]^T \in \mathbb{R}^n$ that is a solution of the system is a possible "operation point" for the modeled application.

The outlined approach is efficient, in fact the simplification of the simultaneous equations can be achieved using well known techniques.

5. Application of the Model. We show how the presented model can be applied to a real application (see Fig. 5.1), a rendering pipeline. The first stage requests the rendering of a sequence of scenes while the second renders each scene (exploiting the PovRay rendering engine), interpreting a script describing the 3D model of objects, their positions and motion. The third stage collects images rendered by the second one, and builds Groups Of Pictures (GOP), that are sent to the fourth stage, performing DivX compression. The last stage collects DivX compressed pieces and stores them in an AVI output file.

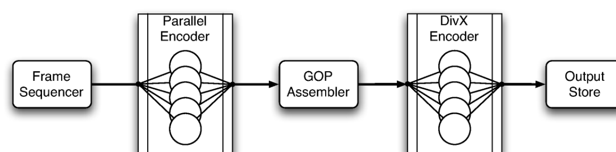

Fig. 5.1. *Graph of the render-encode application*

For GOPs of 12 pictures, the performance model for our test application is (we eliminated execution rates for data-dependent computations):

$$\begin{aligned} C_{1e} = C_{1o} = C_{2i} = C_{2o} = C_{3i} = 12 \cdot C_{3o} = \\ = 12 \cdot C_{4i} = 12 \cdot C_{4o} = 12 \cdot C_{5i} \end{aligned}$$

and has one degree of freedom.

5.1. Convergence to Steady State. We start showing that the application behavior actually tends to steady-state.

Figure 5.2 shows performance features taken from a real execution of the test application on a Blade cluster consisting of 32 computing elements, each equipped with an Intel Pentium III Mobile CPU at 800MHz and 1GB of RAM, interconnected by a switched Fast Ethernet dedicated network. The application was configured to exploit 20 machines in the render computation, and one machine for each remaining node.

Performance features are measured as in (4.2), i. e. averaging the number of performed computations on the duration of the execution. The top diagram shows the performance of the Render and the GOP Assembler nodes, which operate on frames, while the bottom diagram shows the Encoder and Collector nodes, which operate on GOPs. The similarity of the curves in the left and the right diagrams shows empirically that Prop. 4.2 is satisfied not only at the steady-state, but also during the finite computation, as soon as buffers are filled (curves in the same diagram are related by a factor of 1, while between the two diagrams there is a scaling factor of 12).

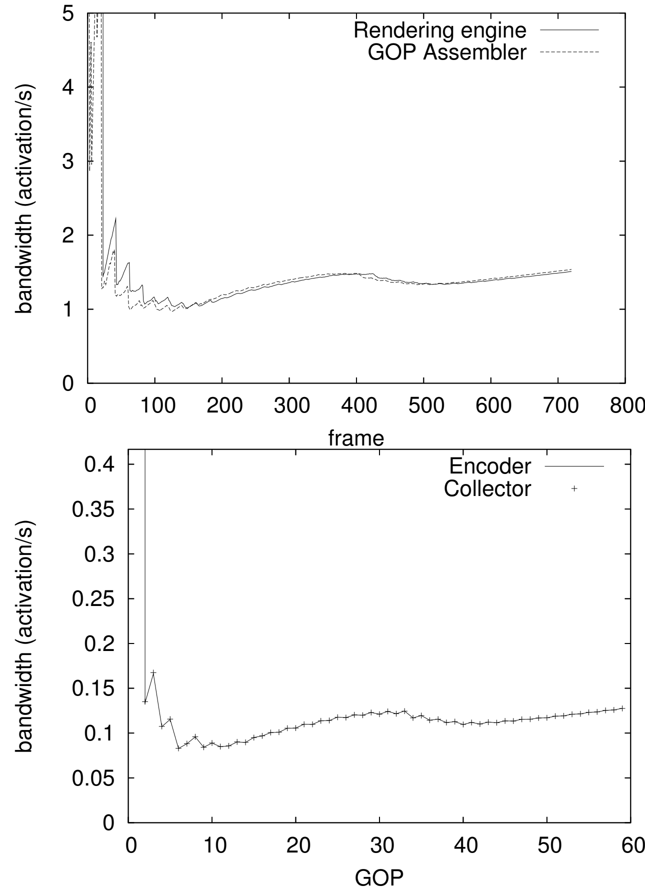

Fig. 5.2. *Convergence to steady-state of averaged performance features*

Moreover, Fig. 5.2 shows that the averaged computation rates stabilize during the computation, allowing us to adopt a steady-state model to approximate the actual application run.

5.2. From Desired Performance to Resource Requirements. Typically, if someone is facing a problem by means of HPC tools, he has clear in mind some sort of performance requirement for his application. This can be expressed in different forms, e.g. completion time, computation rate, response time, etc. In our framework we express requirements as bounds on computation rates. That is the most natural way dealing with stream parallelism. This means that, if the problem is expressed in different terms, some sort of preliminary transformation should be applied (e.g. study the initial transient length to relate completion time to computation rate, or use the Little's Law to translate response time requirements in computation rate ones).

Suppose that we require 1 frame/s (the constraint is expressed by $C_{5i} \geq \dfrac{1}{12}$, because each input for C_5 is composed by 12 frames). Applying the performance model we derive required computation and transfer rates for each computation and communication.

These values, paired with program annotations (see Tab. 5.1) on the weight of computation or communication (e.g. MFLOP per task/MB transferred to/from memory and message size, respectively) can be used to derive requirements that the resources must fulfill in order to meet the performance requirements on the application.

For instance, we can show the requirement for stream $S_2 = C_{2o}$. Since it is required to carry 1.19MB messages with at least rate 1/s, a link of 9.5 Mbit/s is sufficient. Likewise, the test application will never scale above 10 frames/s with a 100 Mbit/s network, and needs to be redesigned, if we want to reach higher performances.

Computational requirements are handled in the same way. The performance model solution gives, for each computation, the minimum required execution rate. Then we need an invertible performance model for

Table 5.1
Deployment annotations for the example application.

Component	C_1	C_2	C_3	C_4	C_5
Processor	i686	i686	i686	i686	i686
Memory (MB)		64	256	64	
CPU Work		3307		52	
Mem. Work		302		104	

Connector	S_1	S_2	S_3	S_4
data type	param	pic	GOP	zip
data size	54B	1.19MB	14.24MB	2MB

each atomic computation that, given the required execution rate, produces the resource requirements. This is essential in an execution environment in which resources are not known in advance.

The model presented in [22] suits our needs. We can associate to each computation a weight, represented by a pair of values $w = (w_{MFLOP}, w_{MB})$, specifying the number of floating point operations (expressed in MFLOP) and the data transferred to/from main memory (expressed in MB) per activation. Resource power is described by the pair $p = (p_{MFLOP/s}, p_{MB/s})$, and execution time is therefore estimated as $t(p, w) = \frac{w_{MFLOP}}{p_{MFLOP/s}} + \frac{w_{MB}}{p_{MB/s}}$.

This model can be employed also to find appropriate parallelism degree for parallel computation nodes. We, in fact, can relate $t(p, w)$ for an aggregate resource $p = [p_1, \dots, p_k]$ to the performance of the code on single resources $t(p_i, w)$.

Assuming perfect speedup, we obtain:

$$t(p, w) = \Big(\sum_i t(p_i, w)^{-1}\Big)^{-1}$$

In this way we can derive, for each computation node, matching resource requirements. These will concern single resources for sequential nodes, and aggregate ones for parallel nodes.

Results commented. In Fig. 5.3, two mappings (top on an homogeneous cluster, bottom with heterogeneous resources) for the same constraint are displayed. The first thing to note is that, even if the heterogeneous run has more variance in achieved bandwidth, the average bandwidth is comparable with the homogeneous one. This provides evidence that the employed performance model correctly handles heterogeneous sets of resources, determining the correct parallelism degree. The good performance in heterogeneous run (its completion time is even shorter than the one for homogeneous run) is explained by the fact that the model can match computation requirements with suitable resources, i. e. schedule memory bound computations (e.g. encoding) on machines with faster memory, and FPU bound ones (e.g. rendering) on machines with faster FPU.

The obtained results are as expected: the mapping computed using the performance model fulfills the constraint, at the beginning and most of the time of the application run. This occurs because, in order to build our model, we sampled the achieved performance on the first frames of the movie, but the application workload slightly changes with the evolution of the movie. This is evidenced by the smoothed bandwidth curve, that has the same course in the two experimental settings: the workload is heavier around 100s and 300s, while it is lighter in the middle and at the end.

6. Conclusions and Future Work. In this work we described an analytical approach to map a class of applications on a Grid. These applications interact through streams of data, processed by several autonomous software components, either sequential or parallel. We presented a steady state performance model for these applications and we applied it to a case study, a rendering pipeline of sequential and parallel components. The model was exploited to predict a program behavior at run-time. Then we showed a general methodology to devise a correct initial mapping for the application, driven by specified QoS levels. At last, we showed the results of our mapping methodology with the presented application, and we discussed the results of the mapping and the execution on homogeneous and heterogeneous sets of resources. We obtained good results in both cases. The application was correctly mapped and the QoS requirement respected with a small error.

Analytical [35, 19, 36, 20, 18, 29] and structural performance models [25, 14, 8] discussed in Sect. 2 need the full knowledge of the target platform to derive performance measures. Therefore, to compare results of different mappings, they must be evaluated multiple times. Our approach decouples the modeling

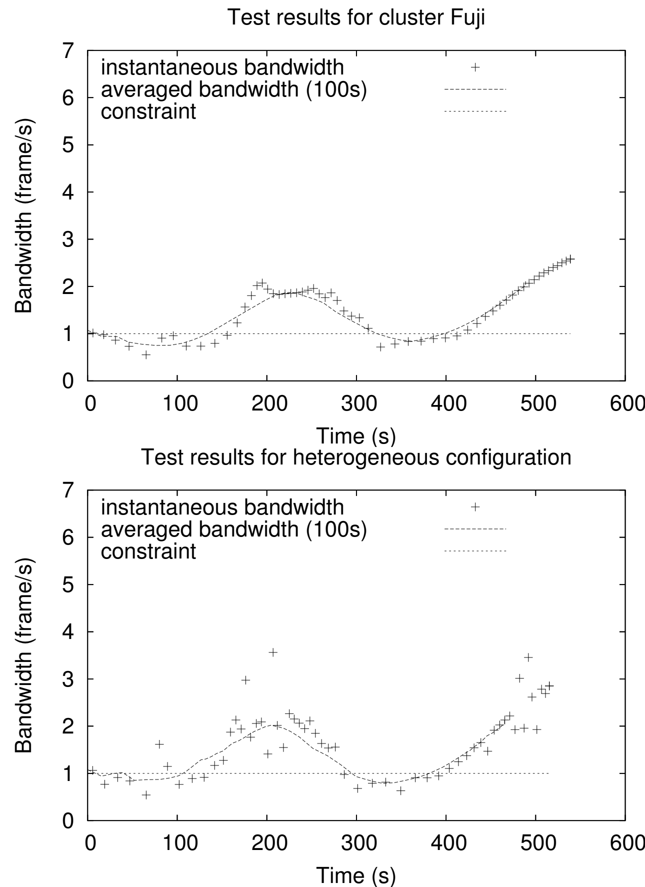

FIG. 5.3. *Two executions of the test application: top) homogeneous clusters of Athlons XP 2600+, down) set of heterogeneous resources (9 P4@2GHz, 1 Athlon XP 2800+, 1 P4@2.8GHz).*

of the application performance from the target platform, allowing us to evaluate the model once to derive enough information to drive the mapping process. Trace-based approaches [34, 26] are used to overcome the limitations of previously discussed approaches, but they are not compositional. Therefore they must be applied from scratch to every new application, even if it is built from the same set of components.

All those models and the presented one share an assumption on the behavior of the applications: computation executions must be independent from the actual values of the input set. Otherwise, two executions of the same application would be not comparable (this is called ergodicity for stochastic models). For applications that do not meet this requirements, the best solution is to resort to runtime adaptation.

The presented approach is not perfect. The initial mapping can be considered a good “hint” to start the execution of an application on a Grid. The dynamic changes in resources during the execution can not be easily included in launch-time strategies. Our approach must be coupled with rescheduling strategies at runtime to solve such problems. Our future work is going in this direction. The presented steady state model can be exploited at run-time to adapt the behavior of components to changes in resource performances. In this way, it should be possible to fulfill the QoS requirements during the whole execution of the application.

REFERENCES

[1] M. ALDINUCCI, A. PETROCELLI, E. PISTOLETTI, M. TORQUATI, M. VANNESCHI, L. VERALDI, AND C. ZOCCOLO, *Dynamic reconfiguration of grid-aware applications in ASSIST*, in Proc. 11th Euro-Par Conference, Lisboa, Portugal, Aug. 2005.

[2] P. AMMIRATI, A. CLEMATIS, D. D'AGOSTINO, AND V. GIANUZZI, *Using a structured programming environment for parallel remote visualization.*, in Proc. 10th Euro-Par Conference, Pisa, Italy, Sept. 2004.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, *Models and issues in data stream systems*, in Proc. 21st ACM-SIGMOD-SIGACT-SIGART Symposium on Principles of database systems (PODS'02), Madison, USA, 2002, pp. 1–16.

[4] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi, *SkIE: a heterogeneous environment for HPC applications*, Par. Comp., 25 (1999), pp. 1827–1852.

[5] S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni, *Model-Based Performance Prediction in Software Development: A Survey*, IEEE Trans. on Software Engineering, 30 (2004), pp. 295–310.

[6] C. Banino, O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert, *Scheduling Strategies for Master-Slave Tasking on Heterogeneous Processors platforms*, IEEE Trans. on Parallel and Distributed Systems, 15 (2004), pp. 319–330.

[7] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert, *Steady-State Scheduling on Heterogeneous Clusters: Why and How?*, in Proc. of 18th International Parallel and Distributed Processing Symposium (IPDPS 04) (IPDPS'04), April 2004.

[8] A. Benoit, M. Cole, S. Gilmore, and J. Hillston, *Scheduling Skeleton-Based Grid Applications Using PEPA and NWS*, The Computer Journal, 48 (2005), pp. 369–378.

[9] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and R. Wolski, *The GrADS Project: Software Support for High-Level Grid Application Development*, Int. J. of High Performance Computing Applications, 15 (2001), pp. 327–344.

[10] D. Bertsimas and D. Gamarnik, *Asymptotically optimal algorithm for job shop scheduling and packet routing*, Journal of Algorithms, 33 (1999), pp. 296–318.

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

[12] M. Coppola and M. Vanneschi, *High-Performance Data Mining with Skeleton-based Structured Parallel Programming*, Par. Comp., Sp. Iss. on Parallel Data Intensive Computing, 28 (2002), pp. 793–813.

[13] M. Danelutto, M. Vanneschi, C. Zoccolo, N. Tonellotto, R. Baraglia, T. Fagni, D. Laforenza, and A. Paccosi, *HPC Application execution on Grids*, in FGG: Future Generation Grid, CoreGRID, Springer, 2006.

[14] M. Dìaz, B. Rubio, E. Soler, and J. M. Troya, *SBASCO: Skeleton-based Scientific Components*, in Proc. of 12th Euromicro Conference on Parallel, Distributed, and Network-Based Processing (PDP'04), A Coruña, Spain, February 2004.

[15] W. Du and G. Agrawal, *Language and compiler support for adaptive applications*, in Proc. 2004 ACM/IEEE Conference on Supercomputing (SC'04), Pittsburgh, USA, Nov. 2004.

[16] N. Dumitrascu, S. Murphy, and L. Murphy, *A Methodology for Predicting the Performance of Component-Based Applications*, in Proc. of 8th International Workshop on Component-Oriented Programming (WCOP 03), Darmstadt, Germany, July 2003.

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

[18] S. Gilmore, J. Hillston, L. Kloul, and M. Ribaudo, *Software performance modelling using PEPA nets*, in Proc. of 4th International Workshop on Software and Performance (WOSP 04), New York, NY, USA, 2004, ACM Press, pp. 13–23.

[19] K. Kant, *Introduction to Computer System Performance Evaluation*, McGraw-Hill, 1992.

[20] P. J. B. King and R. Pooley, *Derivation of Petri Net Performance Models from UML Specifications of Communications Software*, in Proc. of 11th International Conference on Computer Performance Evaluation: Modelling Techniques and Tools (TOOLS 00), London, UK, 2000, Springer-Verlag, pp. 262–276.

[21] H. Kuchen, *A Skeleton Library*, in Proc. 8th Euro-Par Conference, London, UK, Aug. 2002.

[22] A. Litke, A. Panagakis, A. D. Doulamis, N. D. Doulamis, T. A. Varvarigou, and E. A. Varvarigos, *An advanced architecture for a commercial grid infrastructure.*, in European Across Grids Conference, M. D. Dikaiakos, ed., vol. 3165 of Lecture Notes in Computer Science, Springer, 2004, pp. 32–41.

[23] L. Marchal, Y. Yang, H. Casanova, and Y. Robert, *A realistic network/application model for scheduling divisible loads on large-scale platforms*, in Proc. of 19th International Parallel and Distributed Processing Symposium (IPDPS 05) (IPDPS'05), April 2005.

[24] J. Ray, N. Trebon, R. C. Armstrong, S. Shende, and A. D. Malony, *Performance Measurement and Modeling of Component Applications in a High Performance Computing Environment: A Case Study*, in Proc. of 18th International Parallel and Distributed Processing Symposium (IPDPS 04), Santa Fé, USA, April 2004.

[25] J. Schopf, *Structural prediction models for high-performance distributed applications*, in Proc. of the Cluster Computing Conference (CCC'97), Atlanta, USA, March 1997.

[26] L. J. Senger, M. J. Santana, and R. H. C. Santana, *Using Runtime Measurements and Historical Traces for Acquiring Knowledge in Parallel Applications*, in Proc. of the 2004 International Conference on Computational Science (ICCS 04), M. Bubak, G. D. van Albada, P. M. Sloot, and J. J. Dongarra, eds., vol. 3036 of Lecture Notes in Computer Science, Kraków, Poland, June 2004, Springer Verlag, pp. 661–665.

[27] M. Sitaraman, G. Kulczycki, J. Krone, W. F. Ogden, and A. L. N. Reddy, *Performance specification of software components*, in Proc. of the 2001 Symposium on Software Reusability (SSR 01), Toronto, Ontario, Canada, 2001, ACM Press, pp. 3–10.

[28] C. U. Smith, *Performance Engineering of Software Systems*, Addison-Wesley, 1990.

[29] B. Spitznagel and D. Garlan, *Architecture-Based Performance Analysis*, in Proc. of 10th International Conference on Software Engineering and Knowledge Engineering (SEKE 98), Y. Deng and M. Gerken, eds., 1998, pp. 146–151.

[30] N. Trebon, A. Morris, J. Ray, S. Shende, and A. Malony, *Performance Modeling of Component Assemblies with TAU*, in Proc.of CompFrame 2005, Atlanta, USA, June 2005.

[31] S. Vadhiyar and J. Dongarra, *Self Adaptability in Grid Computing*, Concurrrency and Computation: Practice and Experience, 17 (2005), pp. 235–257.

[32] A. J. C. van Gemund, *Symbolic Performance Modeling of Parallel Systems*, IEEE Trans. on Parallel and Distributed Systems, 14 (2003), pp. 154–165.
[33] M. Vanneschi, *The programming model of* ASSIST*, an environment for parallel and distributed portable applications*, Par. Comp., 28 (2002), pp. 1709–1732.
[34] F. Vraalsen, R. A. Aydt, C. L. Mendes, and D. A. Reed, *Performance Contracts: Predicting and Monitoring Grid Application Behavior*, in Proc. of 2nd International Workshop on Grid Computing (GRID 01), London, UK, 2001, Springer-Verlag, pp. 154–165.
[35] L. G. Williams and C. U. Smith, *PASA(SM): An Architectural Approach to Fixing Software Performance Problems*, in Proc. of 28th International Computer Measurement Group Conference, Reno, Nevada, USA, 2002, pp. 307–320.
[36] C. M. Woodside, J. E. Neilson, D. C. Petriu, and S. Majumdar, *The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software*, IEEE Trans. on Computer, 44 (1995), pp. 20–34.
[37] J. Xu, A. Oufimtsev, M. Woodside, and L. Murphy, *Performance modeling and prediction of enterprise javabeans with layered queuing network templates*, in Proc. of the 2005 Conference on Specification and Verification of Component-based Systems (SAVCBS 05), New York, NY, USA, 2005, ACM Press.
[38] A. Zhang, Y. Song, and M. Mielke, *NetMedia: Streaming Multimedia Presentations in Distributed Environments*, IEEE MultiMedia, 9 (2002), pp. 56–73.

Edited by: Pasqua D'Ambra, Daniela di Serafino, Mario Rosario Guarracino, Francesca Perla
Received: June 2007
Accepted: November 2008