Università di Pisa

Scuola di dottorato in Ingegneria
"Leonardo da Vinci"
Corso di Dottorato di Ricerca in
Ingegneria dell'Informazione

Universität Dortmund

Institut für Roboterforschung
Abteilung Informationstechnik

Ph.D. Thesis

Component Performance Modeling and Scheduling Strategies on Grids

Candidate

Nicola Tonellotto

Supervisor
Prof. Ing. Luca Simoncini

Supervisor
Dott. Domenico Laforenza

Supervisor
Prof. Dr.-Ing. Uwe Schwiegelshohn

2008

Abstract

With a Grid, networked resources, e.g. desktops, servers, storage, databases, even scientific instruments, could be combined to deploy massive computing power wherever and whenever it is needed most. In the past years, High Performance Computing (HPC) applications have taken full benefit from this power, but to completely exploit the potential of Grids, a whole new set of requirements must be handled: security, resource discovery, access and management, application deployment, heterogeneity, dynamicity, interoperability, quality of services (QoS), etc. To address such problems recently a new abstraction has been introduced: the *component*. Component technologies have been successfully adopted in sequential programming and distributed computing, and currently several component-based programming models have been proposed as a solution to the major problems arising when programming Grids.

In this thesis, three key requirements for Grid application development software are investigated: the need of a performance model for components, in order to derive the performance of Grid applications; the need of a performance contract for components, in order to derive the requirements for the execution of component-based applications on heterogeneous platforms starting from user-level performance requirements; and the need of mapping strategies for tightly-coupled applications developed using the component paradigm, with a particular focus on the impact of communication requirements. The contributions of this thesis are structured as follows.

- A *compositional performance model* for components interacting through streams of data is proposed. This model is derived from an analytical model of the dynamic behavior of sequential and parallel components.
- A definition of *performance contract* for component-based applications is proposed. This definition, based on the results on the steady state limit for the performance model, is viable in general for hierarchical component models with asynchronous one-way communications.
- A polynomial-time *propagation algorithm* for component contracts is proposed. Starting from a performance model and a performance contract, it allows to compute the performance parameters that every component must satisfy, so that the program, as a whole, will fulfill the performance contract at runtime.
- Two launch-time scheduling heuristics to allocate component-based applications are proposed. The applications are described by *task interaction graphs* whose nodes and edges are labeled according to the QoS requirements derived through the contract

resolution algorithm. The goal of such heuristics is to obtain feasible mappings, dealing with computational and communication requirements. The first one, the Wide Area Scheduling hEuristics (WASE) targets hierarchically structured Grids exploiting qualitative information on the application structure, while the second one, the QoS List Scheduling hEuristics (QLSE), targets unstructured global Grids trying to exploit quantitative information on the application structure.

Acknowledgements

It has been a great privilege to have an international jointly supervised PhD dissertation. My thanks go to Dr. **Domenico Laforenza** (ISTI-CNR), Prof. **Luca Simoncini** (University of Pisa) and Prof. Dr.-Ing. **Uwe Schwiegelshohn** (University of Dortmund) who have made it possible. A special thank goes to Dr.-Ing. **Ramin Yahyapour**, who offered me not only writing and academic guidance, but also general support and encouragement. I am also deeply indebted to **Ranieri Baraglia** and Dr. **Corrado Zoccolo**, for their continuous support and fruitful discussions. They have given me a lot of useful ideas, many of which are contained within this work. I am also very grateful to many other colleagues in the High Performance Computing Laboratory in Italy and the Institute for Robotics Research in Germany: Dr. **Fabrizio Silvestri**, **Diego Puppin**, **Claudio Lucchese**, **Patrizio Dazzi**, **Antonio Panciatici**, **Marco Pasquali**, Prof. **Salvatore Orlando**, **Renato Ferrini**, **Raffaele Perego**, **Lars Schley** and Dr. **Jiadao Li**. Many thanks go to Prof. **Marco Vanneschi**, Prof. **Marco Danelutto** and Dr. **Massimo Coppola** of the Computer Science Department of the University of Pisa which supported the early stage of my work as members of the *Grid.it* project. I would like to express my gratitude to the CoreGRID Network of Excellence, which made possible my joint PhD, and its members, in particular I want to thank **Philipp Wieder**, Dr. **Wolfgang Ziegler** and **Oliver Wäldrich**, which help me with a fertile discussion ground during many meetings and conferences. Many thanks also go to my friends **Antonio**, **Monica**, **Simona**, **Elena**, **Gabriele**, **Alessandro**, **Davide**, **Diego**, **Manolo**, **Francesca O.**, **Andrea**, **Francesca V.**, **Franco**, **Andrea K.** and, in particular, **Laura** and **Mirella**, who supported me with their love and friendship during these years.

εἰ δὲ φαίνεται θεασαμένοις ὑμῖν, ὡς ἐκ τοιούτων ἐξ ἀρχῆς ὑπαρχόντων, ἔχειν ἡ μέθοδος ἱκανῶς παρὰ τὰς ἄλλας πραγματείας τὰς ἐκ παραδόσεως ηὐξημένας, λοιπὸν ἂν εἴη πάλτων ὑμῶν [ἢ] τῶν ἠκροαμένων ἔργον τοῖς μὲν παραλελειμμένοις τὴς μεθόδου συγγνώμην τοῖς δ΄ εὑρημένοις πολλὴν ἔχειν χάριν.

ΠΕΡΙ ΤΩΝ ΣΟΦΙΣΤΙΚΩΝ ΕΛΕΓΚΩΝ
Ἀριστοτέλης

If, then, it seems to you after inspection that, such being the situation as it existed at the start, our investigation is in a satisfactory condition compared with the other inquiries that have been developed by tradition, there must remain for all of you, or for our students, the task of extending us your pardon for the shortcomings of the inquiry, and for the discoveries thereof your warm thanks.

ON SOPHISTICAL REFUTATIONS
Aristotle

Contents

List of Figures

List of Tables

List of Algorithms

1

Introduction

Scientists are always facing increasingly complicated problems and a single computer is not enough for the calculations they want to do. Even the computers leading the world in terms of processing capacity (*supercomputers*) have shown their limits in term of the problems they are able to solve (e.g. problem size and speed). In the nineties, a new type of distributed computing paradigm was proposed. This paradigm, called *metacomputing* [1] tried to overcome the lack of computing power by linking up supercomputer centers with what were, at the time, high speed networks. However supercomputers are expensive and a relatively small number of companies and research centers were able to have one of them. Building a metacomputer was an extremely complex task, and very few scientists were able to access one and to fully exploit its capabilities.

Fortunately, the last twenty years have seen a considerable increase in computer and network performance. In few years, it became clear that it was possible to reach impressive computing power simply by linking together several *commodity-off-the-shelf* homogeneous computers hosted in research center rooms. The idea to exploit the computing power of homogenous workstations interconnected through a local area network gave birth to the *cluster of workstations* (COW). This idea proved successful and was further investigated. The restriction of local interconnections between the resources was relaxed and workstations in different academic and research institutions were connected, crossing their respective borders in a *network of workstations* (NOW). Next, the restriction of resource homogeneity was relaxed, and thus far more resources could be pooled, with different interconnection characteristics: the dream is to exploit idle cycles of the computers connected to the Internet. This global scale vision of distributed computing is often referred as *Grid computing*.

Grid computing has specific requirements due to its inherent largely distributed and heterogeneous nature. Research efforts first focused on the access to physical resources by providing tools for search, reservation and allocation of resources, and for the construction of *Virtual Organizations* (VOs) [2]. Managing the Grid resources is a first necessary step for taking advantage of the Grid infrastructure, and the second step is to offer adequate development models and environments. A new research area therefore emerged, focusing on programming models and tools to efficiently program Grid applications.

Traditionally, the target for parallel programming has been limited to some specialists with interest in, and knowledge of, the problem being solved. They developed their

parallel algorithms often from scratch, using low level programming tools, for the lack of powerful parallel programing tools and, of course, performance. Parallel programming tools were not effective from the point of view of software companies. The first commercial prototypes of such tools were introducing too much overhead with respect to low level programming. The subsequent widespread adoption of low cost clusters and networks of workstations changed this scenario, and the need of high level parallel programming tools became more urgent. The adoption of *structured* parallel languages [3] permitted the reuse of sequential code to build parallel algorithms, which could in turn be composed to build a parallel application. This programming paradigm was able to raise the level of abstraction provided to programmers as well as to provide separation of concerns between application and system programmers. With the advent of Grid computing a whole new set of requirements is introduced: security, resource discovery, access and management, application deployment, heterogeneity, dynamicity, interoperability, quality of services (QoS), etc.

To address such problems, a new abstraction has recently been introduced: the *component* [4]. A component is a reusable piece of code which can be invoked according to a well-defined interface. Structured parallel programming and component technologies are two non mutually exclusive ways of structuring code in order to simplify the development of complex parallel applications, starting from simpler building blocks.

Component technologies have been successfully adopted in sequential programming and distributed computing. Currently several component-based programming models have been proposed to overcome the major problems arising when programming Grids.

1.1 Grids

The popularity of the Internet and the availability of powerful computers and high-speed networks as low-cost commodity components have led to the possibility of using geographically distributed and multi-owner resources to solve large-scale problems in science, engineering and commerce. Recent research on such topics has led to the emergence of a new computing paradigm known as *Grid computing*. The term Grid was chosen as an analogy to a power Grid that provides consistent, pervasive, dependable transparent access to electricity, irrespective of its source [5]. The concept of Grid computing started as a project to link geographically dispersed supercomputers [6], but has now grown far beyond that original intent. The use of the Grid infrastructure can benefit many applications, including collaborative engineering, data exploration and distributed supercomputing [7].

It is possible to distinguish different general layers in a Grid architecture [2]: at the lowest level the local resources (computers, networks, sensors, databases). On top of it, the middleware is built, in order to hide the heterogeneous nature of underlying resources and to provide a small set of low-level tools to access and manage such resources. The Globus Toolkit[1] is the "de facto" standard middleware to build Grid infrastructures.

These two layers just define the running environment of the true beneficiaries of the Grid infrastructures, the applications. The next layer of the Grid architecture must then

[1] `http://www.globus.org/toolkit`

provide a set of high-level tools to allow the development and the execution of applications that can not be run on different architectures. Grid applications are currently developed exploiting directly the Grid middleware. Overall, the tools and APIs of the middleware burden directly the programmers with the control over Grid resources, process and communication management. Currently, only a few classes of applications are successfully executed on Grids: minimal communication applications (embarrassingly parallel computations), staged applications (get inputs/computes data/visualize outputs), single resource applications (get something from A, do something at B). In coming years, applications will use Grids in more sophisticated ways, adapting to dynamic configurations of resources and performance variations to achieve the goals of autonomic computing [8]. To reach these goals, some key research areas that will provide the building blocks for future Grids are:

- **Adaptive applications**: the development of software which is self-optimizing, self-configuring, self-healing and self-protecting will allow programs to adapt to the dynamic performance that can be delivered by Grid resources. Moreover, as the Grid infrastructure grow in size and complexity, it will need a robust and flexible architecture that can easily adapt to change.
- **Grid programming environments**: high-level programming environments hiding the complexities of the Grid infrastructures will be the key factor to increase the productivity of Grid environments. Application developers will concentrate on the problem being solved rather than on the issues introduced by the target architecture. New compile-time and runtime tools for applications are needed. Such tools must interact with resource discovery and selection mechanisms to best target their programs, and to migrate them during execution in case of performance failures.
- **New technologies**: new access devices and new information sources will be integrated in the Grid infrastructure. Mobile systems such as cell phones or PDAs will allow ubiquitous access to Grid infrastructures, as well as new information sources. Moreover, new sensors and sensor nets will provide an immense source of data, creating a new level of potential for scientific applications.
- **Grid economies**: the Grid is growing to be a global-scale complex system. In the future, this system will require policies, structure and economies to maintain stability and provide efficient performance.

1.2 Components

Component-based programming takes the idea of using construction blocks and elements for building manufactured products, and applies it to software[9]. A *component* is a reusable program building block that can be combined with other components in the same or other computers in a distributed network to form an application. Components can be seen as an extension of objects: they are aiming, like objects, at code reuse, but focusing also on the issues of software interoperability, providing language independence, compiler independence, and seamless access to distributed object resources.

Components rely on the decoupling of interfaces and implementations: interfaces are defined in a special purpose Interface Definition Language (IDL), and are bound to implementations (written in one of the supported languages) by means of an IDL compiler.

Components are composed according to a *composition model* (in order to build a more complex application). In this way the programmer is relieved from programming the low-level interactions of different components; rather it is responsibility of the component programming framework to implement it efficiently.

Component-based programming is good to use for the following three reasons:

- **Encapsulation**: components are seen as black-boxes, which define functionalities offered and functionalities required through interfaces; they rely on the decoupling of interfaces and implementations.
- **Composition**: components are composable; this facilitates the design of complex systems by simply assembling functional software units, and hierarchical models offer abstractions of composed sub-systems.
- **Description**: assemblies of components can be analytically described with a standard Architectural Description Language (ADL). The description of components and component-based applications can also include performance and deployment information, to be used during the deployment of the application and at runtime.

The software development process benefits from this approach. The clear separation between interfaces and implementations allows the basic components to be developed by specialists with knowledge in the functionalities offered and programming skills to certify the quality of the implementation. Because of this, the complex components can be assembled by composing other components. The interactions between software units are completely specified by the components' interfaces, and an application developer can not just select the right components and connect their interfaces to build a complete application. The modularity and encapsulation of components make them perfect candidates for the execution of applications in distributed environments, which can be easily carried out by the component framework. Moreover, the description of components and applications allows to build automatic mechanisms to deal with deployment and runtime issues typical of distributed computing.

1.3 Statement of the Problem

It is becoming increasingly clear that existing Grid middleware, such as the Globus Toolkit and associated software, does not provide the required support to enable easy and reliable application construction and execution in a Grid environment [10]. Impressive applications have been developed, but only by teams of specialists. Entirely new approaches to software development and programming are required for Grid computing to become broadly accessible to non specialist scientists, engineers, and other problem solvers.

New Problem Solving Environments (PSEs) are required, to automatically manage the Grid issues and to provide to programmers the facilities to hide the Grid infrastructure. To implement this *invisible Grid* [11], new programming environments, models, and tools are required. Recently, some promising Grid application development softwares have been prototyped. The GrADS[2] project has implemented a new program development

[2] `http://hipersoft.rice.edu/grads/`

and execution structure, encapsulating applications in *configurable object programs* to be dynamically mapped on Grid resources, and relying on specific *performance contracts* to steer the configuration of applications at runtime to achieve expected performance. The *Grid.it* project has improved the structured parallel programming environment ASSIST[3], whose design is aimed at raising the level of abstraction in Grid programming exploiting structured high performance components and implementing dynamic resource discovery and selection as well as dynamic adaptation of applications.

However, such Grid software needs a number of auxiliary technologies to support complex applications development and execution. Component-based programming is currently the most promising paradigm for programming complex systems, breaking them in smaller and simpler pieces: it is well suited to efficiently face the new challenges in terms of programmability, interoperability, code reuse and efficiency that mainly derive from the features that are peculiar to Grids. Nevertheless, the integration of this programming model with the Grid infrastructure and its adoption from the scientist communities still faces several research challenges, including:

- component interactions must be defined precisely and in such a way that complex, multidisciplinary applications can be constructed by the composition of building block components, possibly obtained by suitably wrapping existing code;
- performance/cost models must be defined to allow the development of tools for reasoning about components and their deployment;
- suitable tools must be designed which will allow for seamless integration of the component runtime environment with the Grid middleware;
- existing algorithms must be adapted to work with component applications and their performance models; if necessary, new solutions must be proposed.

Clearly, for this execution scenario to work, we must have a reasonable performance model and mapping strategies for each application. Performances models are needed to correctly drive the resource selection phase at launch time. At runtime, performance models can be automatically managed by the framework to selected new resources where components should be migrated to fulfill the application QoS. Mapping strategies are needed to manage the new complex applications and to handle their execution on the target Grids, facing issues like heterogeneity in resources architecture, power and data formats and limited, varying communication bandwidths.

1.3.1 Purpose of the Study

This thesis investigates three key requirements for Grid application development softwares:

- the need for a performance model for components, in order to derive the performance of Grid applications;
- the need for a performance contract for components, in order to derive the requirements for the execution of component-based applications on heterogeneous platforms starting from user-level performance requirements;

[3] `http://www.di.unipi.it/Assist`

- the need for mapping strategies for tightly-coupled applications developed using the component paradigm, with a particular focus on the impact of communication requirements.

The adoption of the component paradigm to develop Grid applications allows different programmers to implement specific-purpose components. The programmer's expertise in the implementation should allow him to describe in detail the performance characteristics of the specialized component. A compositional performance model is then envisioned, in which performance features of different components can be composed to automatically build the performance model of the whole application.

Often application users do not have the knowledge or expertise needed to fully understand the structure and the performance of applications. Typically, Grid users express their requirements on the execution of an application expressing an high-level goal (e.g. I want an overall throughput of x data/sec, I want the application to execute in less than 2 hours). In doing so they stipulate with the application execution framework a sort of "contract" that the application should respect at runtime. This contract may include rewards and penalties for the fulfillment of the requirements. The runtime environment of the application (execution support and system middleware) is in charge of controlling the fulfillment of the contract exploiting the performance features of the model.

Composing components to build applications can easily drive to the building of very complex programs, which have several computing components interconnected with structured and unstructured communication patterns. However, the mapping of such applications on Grids is difficult. The general problem of mapping a set of tasks on distributed resources is known to be NP-complete and even if some special instances are solvable in polynomial time, the dynamicity of the Grid can make an optimal or quasi-optimal solution useless in short time. Instead of elaborating very complex heuristics to try to optimally map components on a Grid, it is preferable to devise a launch-time scheduling heuristics able to find a mapping to guarantee the QoS required by the end-user (a *feasible* mapping). With optimal mappings, a dynamic change of resource characteristics may result in a probable loss of optimality, so the time and computational efforts in deriving them might not be justifiable. The main current approach to dynamic changes management is to rely on dynamic rescheduling mechanisms. Recalculating an optimal assignment may be again time-consuming, while devising new feasible mapping for just the components violating the QoS constraints may be more reasonable.

This thesis proposes the following contributions:

- A *compositional performance model* for components interacting through streams of data. This model is derived from an analytical model of the dynamic behavior of sequential and parallel components.
- A definition of *performance contract* for component-based applications. This definition, based on the results on the steady state limit for the performance model, is viable in general for hierarchical component models with asynchronous one-way communications.
- A polynomial-time *propagation algorithm* for component contracts. Starting from a performance model and a performance contract, it allows the computation of the per-

formance parameters which every component must satisfy, in order for the program, as a whole, will fulfill the performance contract at runtime.

- Two launch-time scheduling heuristics to allocate component-based applications. The applications are described by *task interaction graphs* whose nodes and edges are labeled according to the QoS requirements derived through the contract resolution algorithm. The goal of such heuristics is to obtain feasible mappings, dealing with computational and communication requirements. The first one, the Wide Area Scheduling hEuristics (WASE) targets hierarchically structured Grids exploiting qualitative information on the application structure, while the second one, the QoS List Scheduling hEuristics (QLSE), targets unstructured global Grids trying to exploit quantitative information on the application structure.

The performance model and contract have been applied to the problem of the deployment onto a Grid platform of an ASSIST applications. This has been done as part of the Italian national project *Grid.it* (Enabling Platforms for High-Performance Computational Grids Oriented to Scalable Virtual Organizations).

The launch-time scheduling heuristics have been studied to evaluate the impact of communication requirements on the mapping of applications on Grids. This work has been done as part of the CoreGRID Network of Excellence (European Research Network on Foundations, Software, Infrastructures and Applications for large scale, distributed, GRID and Peer-to-Peer technologies).[4]

1.3.2 Limitations of the Study

The proposed *performance model* for components interacting through streams of data is an innovative, general approach to the analytical modeling of hierarchical components. Although the model is limited to streams, which are a common communication pattern in high performance and high throughput computing, it is viable for hierarchical component models with asynchronous one-way communications like Fractal [12]. It can be exploited to reason about the dynamics of running components and it is suited for simulation and control environments (e.g. Simulink®[5]).

The proposed *performance contract* is, at the best of the author knowledge, the first approach in Grid computing to a performance contract not focused just on the monitoring and runtime reconfiguration of an application, but also on the management of user-level QoS requirements. The performance model/contract and the execution platform are independent, so it is viable to exploit the application performance characteristics on different models of Grid resources, although their applicability needs further investigation.

The proposed *scheduling heuristics* are not sub-optimal algorithms. Their approximation of the optimal solution is not, in general, quantifiable. This is due to the main assumption that the dynamicity of the Grid will hinder the use of optimal mappings, and thus dynamic rescheduling approaches must be exploited. 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. The proposed approach must be coupled with rescheduling strategies at

[4] `http://www.coregrid.net`
[5] `http://www.mathworks.com/products/simulink/`.

runtime to solve such problems. The presented steady state model can be exploited at runtime to adapt the behavior of components to changes in resource performances [13].

1.4 Summary

The rest of the thesis is organized as follows. In Chapter 2 backgrounds on component technologies and Grid computing are given, positioning the presented work with respect to the state of the art. The dynamic model of components and their steady-state behavior is illustrated in Chapter 3, and in Chapter 4 the steady-state limit is exploited to define the performance contract and the relative contract propagation algorithm is presented and discussed. In Chapter 5 and Chapter 6 the Wide Area Scheduling hEuristics (WASE) and the QoS-constrained List Scheduling hEuristics (QLSE) are presented, respectively. Their performance evaluation is provided. In Chapter 7 the conclusions on the presented work are drawn, and further developments are discussed.

2

Background

This chapter reviews the state of the art of component technologies, component performance model and Grid scheduling. First, it will review the most important component technologies in industry and research, focusing on the component frameworks proposed in Grid computing, followed by an introduction of the Grid concept and a presentation of the topics related to the contents of this thesis. The end of each section will discuss the positioning of this work with respect to the presented topics.

2.1 Components

Traditionally, the target for *parallel programming* has been limited to specialists, with interest in and knowledge of the problem being solved. They often developed their parallel algorithms from scratch, using low-level programming tools, because of the lack of powerful parallel programing tools and, of course, performance. From the point of view of software companies, parallel programming tools were not effective, and their first prototypes introduced too much overhead with respect to low-level programming. Subsequently, low cost clusters and networks of workstations were adopted and the scenario; the need of high level parallel programming tools became more urgent. The adoption of *structured parallel languages* (i.e. skeletons [3, 14]) permitted the reuse of sequential code to implement parallel algorithms, which could in turn be composed to build a parallel application. This programming paradigm was able to raise the level of abstraction available to programmers and to provide separation of concerns between application and system programmers. However, the advent of Grid computing has introduced a completely new set of requirements which must be handled by programming environments: security, resource discovery, access and management, application deployment, heterogeneity, dynamicity, interoperability, quality of services (QoS).

Recently a new software abstraction has been introduced to address these problems: the *component*. The underlying idea is to construct software in the same way as hardware, i.e. by assembling reusable components. The initial concept of component was introduced by McIllroy in 1968 [9]. Since then the notion of component has developed several definitions of a component are grown in the literature [15, 16, 4] because components have many varied forms and characteristics. Today, the commonly accepted definition of component is that of Szyperski [4]:

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

This definition states that a component must have clearly specified interfaces and its implementation must be encapsulated in the component itself, not directly reachable from the environment. This implies that component dependencies must be designed in such a way to be treated as independently as possible. Moreover, to be deployed autonomously, a component must be clearly distinct from its environment and from other components.

Indeed, *object oriented programming* [17] enabled the development of reusable class libraries, such as the Standard Template Library (STL) [18] or Java Foundation Classes (JFC) [19]. Moreover, distributed technologies like the Common Object Request Broker Architecture (CORBA) [20] and Java Remote Method Invocation (JavaRMI) [21] gave the impulse to *distributed object* computing environments.

However, object-oriented methodology never achieved software reuse and composition in the large, mainly because:

- a large number of fine-grained classes are generated during object-oriented modeling;
- these classes are entangled in a web of association, aggregation and generalization relationships;
- the deployment context of a class does not change after compilation, and often the source code is required, making difficult to reuse existing components;
- these technologies mainly target sequential applications, also in distributed environments, and they are completely unaware of the underlying running environment;
- fine-grained aggregation of classes makes well-defined, context-independent interfaces for composition difficult to expose.

The next section will review several technologies and related models for components. For each component model discussed, it is defined what the component is, what the interfaces look like and how components interact and communicate.

2.1.1 Industrial Component Models

Several companies (e.g Microsoft and Sun Microsystems) have recognized the need of distributed computing environments built on top of components. Several commercial standards have been proposed, and today Microsoft's COM/.NET, OMG's CCM and Sun's EJB represent the most common component models used in industry.

COM/.NET

The first proposal for a component model fostered by Microsoft was the Component Object Model (COM)[1]. COM is a binary standard not tied to a particular language, and its goal is to specify how COM components (represented by binary executable files) interact with other objects. In doing so, COM enforces a clear separation between the implementation and the interface parts of an object. In fact, the fundamental entity of COM objects is the *interface*, which represents the access point to the object. Such

[1] `http://www.microsoft.com/com`.

interfaces are externally specified through the Microsoft Interface Definition Language (Microsoft IDL), and the interaction is implemented through stubs and proxies with remote procedure calls.

Moreover, the COM framework implements a set of key services and functionalities to provide interfaces introspection and dynamic interface discovery. The COM framework has evolved and now includes COM+, Distributed COM (DCOM) and ActiveX Controls. COM+ enriches the set of COM services providing new mechanisms for resource pooling, event publication and subscription and distributed transactions. DCOM adds distributed communication capabilities, taking charge of the low-level details of network protocols. ActiveX Controls define standard COM interfaces for compound documents. The COM framework is now being superseded by the .NET technology[2], providing a broader set of services and interoperability, notably through web services.

CCM

The CORBA Component Model (CCM) [22] is an Object Management Group (OMG) specification to build system- and language-independent, distributed, robust, heterogeneous business components. CCM is built on the OMG's CORBA specification and extends its concept of distributed objects. A CORBA component is a black box CORBA object with the following access interfaces: a self-referent interface (allowing introspection), a set of facets (provided interfaces), a set of receptacles (required interfaces), a set of event sources and sinks (implementing asynchronous communications) and a set of configuration attributes. CORBA components are automatically deployed in a *container* and created by a *home*. The container provides services for persistency, security and transactions.

The CCM framework provides tools to build multi-component applications, connecting facets of one component to receptacles of another component. It also provides asynchronous communication patterns: in addition to CORBA's RPC mechanism, it supports event-based communications.

EJB

Enterprise Java Beans (EJB)[3] is a Sun technology that allows the modular construction and management of server-side applications. The EJB specification is built upon *beans*, which are distributed objects incapsulating business code. A bean's whole lifecycle is managed automatically by a *container*, which is also in charge of the management of non functional features like persistency, security and load balancing. Beans are created and managed by a *home* container service, and they can be deployed independently in different EJB containers through specific *deployment descriptors*. Descriptors are XML files containing information about the bean's name, Java interfaces and implementing classes, persistent data and security policies.

However, the programming model of these applications is still client-server, and the interaction mechanism follows the RMI/CORBA approach. Moreover, EJB does not

[2] `http://www.microsoft.com/net`.
[3] `http://java.sun.com/products/ejb/`.

directly address the issue of language interoperability. Although the Java Native Interface library supports interoperability with C and C++, using the Java virtual machine to mediate communication between components would incur an intolerable performance penalty on every inter-component function call.

2.1.2 Academic Component Frameworks

Research communities have proposed different component models and frameworks (e.g. CCA, ASSIST, Fractal). These models are not widely adopted in production environments and they usually focus on a specific research topic (e.g. high performance, interoperability, quality of service, adaptivity), facing a problem unsolved in actual computational frameworks [23].

CCA

The Common Component Architecture (CCA) [24] is a specification of a minimal component architecture for high-performance computing. This specification establishes the basic rules to build components portable between frameworks, specifying a minimal set of interactions between components and the framework and the minimal set of behaviors a component must exhibit. Components are software entities living inside a framework and accessible through client and server ports. A framework is a container for building, connecting and executing components. The component ports and the interactions with the framework are described through the scientific IDL (SIDL), which at the same time provides the basis for the interoperability. SIDL enriches CORBA IDL with high-performance specific data types. Components must be reflective, but the programmer is responsible for the implementation of introspection. Component binding is carried out by Babel [25], a specialized tool which automatically generates the required wrapper code. Ports and their connections are fully dynamic. They can be added, removed and connected at runtime. However, hierarchical composition is not included in the CCA specification.

The CCA specification has been implemented in several projects, each one tailored for specific needs. For example, XCAT [26] implements components as Grid services exploiting the Globus Toolkit services, CCAFFEINE [27] targets at high-performance SPMD computations, exploiting a reduced set of functionalities for message passing and distributed memory, while DCA [28] bases its implementation on MPI to benefit from its high performance and scalability.

ASSIST

ASSIST (A Software development System based on Integrated Skeleton Technology) [29] provides programmers with a structured coordination language. This language can be used to express parallel programs at a very high level of abstraction. The ASSIST coordination language programs are made up of two specific parts. A *module graph* describes how a set of modules, either parallel or sequential, interact with each other using a set of data-flow streams. A set of *modules* implements the nodes of the graph. Sequential modules are basically procedure-like wrappings of sequential code written in C, C++

or Fortran code. Parallel modules are programmed instantiating an ASSIST-specific parallel module.

In addition to the structured coordination language and related tools, ASSIST provides several features, such as external libraries and objects that can be encapsulated in CORBA objects. Moreover tools to compile ASSIST programs into CCM components [30] or even into standard Web Services [31] are available.

Fractal

Fractal is a modular and extensible component model [12] consisting of a generic component model, several implementations (e.g. Julia and AOKell) and a set of tools and reusable components. It relies on the strong decoupling between interfaces (constituting the *membrane* of a component) and implementation. Fractal provides standard APIs to access component information (*introspection*) and to dynamically modify a component configuration (*intercession*). Moreover, standard APIs are included in the model which allows the control of the lifecycle, the attributes and the bindings of a component. A prominent feature of this model is its composition model, allowing the composite components to be built from simpler ones, in such a way that offers a uniform point of view of applications at various levels of abstraction. Another interesting feature of the Fractal component model is the enforcement of the separation of concerns between functional interfaces (granting access to business code) and non-functional ones (devoted to the management of the component).

2.1.3 Grid Component Frameworks

The Grid infrastructure imposes new requirements on component frameworks. On the one hand, coupling component frameworks with structured parallel programming will allow to rise the level abstraction to build complex applications that can really benefit from the Grid infrastructure. On the other hand, these frameworks must extend the set of provided services to manage all the details which programmers must typically deal with when programming Grid applications. The following frameworks are the first research prototypes trying to face these requirements (GridCCM/PadicoTM, GCM, Grid.it, HOC and ProActive).

GridCCM/PadicoTM

Exploiting the concept of parallel objects investigated in several research prototypes (ParDIS [32], PaCO++ [33]), GridCCM [34] is an ongoing project aiming at leveraging the CCM capabilities (support for heterogeneity, open standards and deployment mechanisms) to include high-performance SPMD codes with optimized collective communications. While GridCCM is trying to raise the level of abstraction required to run high-performance applications, PadicoTM is designed to be a portable and efficient runtime environment for computational Grids that allows components to communicate with each other using the available underlying network.

GCM

The Grid Component Model (GCM) [35] is a research effort of the CoreGRID Network of Excellence to define a standard component model for Grid applications. Starting from the FRACTAL component model, the GCM aims to address the new challenges of Grid computing - heterogeneity and dynamicity - in terms of programmability, interoperability, code reuse and efficiency. The main features that a component framework must exhibit to conform to the GCM specification are hierarchical composition of (GCM) components, structured communications, autonomic reconfiguration, functional and non-functional dynamic adaptation, behavioral protocol specifications and deployment on Grid infrastructures.

Several preliminary studies have been conducted to investigate implementations of such features. A new research project (GridCOMP[4]) was funded to build a reference implementation of the GCM.

Grid.it

The ASSIST programming model has been extended with the concept of component in the context of the *Grid.it* project[5][36]. Its model has a recursive, hierarchical definition, which allows the wrapping of applications developed with other frameworks (e.g. CCM, Web Services) in a Grid.it component, mainly thanks to different interaction semantics. The runtime support is implemented on top of a Grid Abstract Machine (GAM) [37], providing the Grid functionalities required by the programming environment as abstract services (resource discovery, management, monitoring, component deployment, execution and wiring, routing of communications). Each Grid.it component has a set of non-functional interfaces that both report at runtime the details and performance of the component, and perform a sequence of dynamic reconfigurations.

Moreover, the ASSIST component framework proposes the concept of *supercomponent*, a graph of basic components automatically managed by a hierarchical structure of controller entities able to enforce the desired performance through dynamic reconfiguration at runtime [38].

HOC

High Order Components (HOCs) [39] are founded on the concept of components as a reusable unit of composition and algorithmic skeleton, to raise the level of abstraction in high-performance computing. A gap exists between Grid and components and is bridged by HOC-Service architecture (HOC-SA) [40], implemented using the Web Service Resource Framework (WSRF) [41]. The most innovative feature of HOC components is their ability to distribute over the network executable code, to tailor the behavior of components on a per-user basis. Such *code mobility* (which not only includes Java, but C++ as well) is implemented through a *code service*, where users can upload custom code, and a remote *code loader*, integrated in HOC-SA services, to download the required code from the code server when needed.

[4] `http://gridcomp.ercim.org/`.
[5] `http://www.grid.it`.

ProActive

The ProActive middleware[6] is a Java library aiming to achieve seamless programming for concurrent, parallel, distributed and mobile computing. It is based on the concept of *active object*, a remotely accessible object (via method invocations) implementing asynchronous and group communications, code migration and automatic deployment.

ProActive has been extended to provide a distributed implementation of the Fractal component model [42]. Thus all the features of the Fractal specification are available in distributed, heterogeneous environments, and the ProActive implementation is in charge of the management of the Grid technical issues.

2.1.4 Component Performance Models

Performance specification of components and their interactions is a basic problem that must be solved to enable software engineers to assemble efficient applications [43]. Moreover, performance modeling is one of the key aspects that needs to be addressed in order to face scheduling/mapping problems in heterogeneous platforms. This 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. These include analytic performance models, symbolic performance models, asymptotic steady-state analysis, structural performance models and trace-based performance models.

Analytic performance models in software engineering make extensive use of UML formalism to describe software component behavioral models [44, 45] and to derive models based on Queuing Networks [46] or Layered Queueing Networks [47] to be exploited in design phase of the lifecycle of software. The same holds for Stochastic Petri Nets [48, 49] and Stochastic Process Algebras [50, 51]. These 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 [52]). A detailed survey of of such models is in [53].

Symbolic performance modeling [54] is a methodology that enables 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 [54] 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 DAGs[7] 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 this approach is that it computes lower bounds of the performance of a program.

The asymptotic steady-state analysis was pioneered by Bertsimas and Gamarnik [55]. This approach has been recently applied to mapping and scheduling problems of parallel applications on heterogeneous platforms [56, 57, 58], in which the analysis is applied to particular classes of parallel applications (divisible load [56], master/slave [58], pipelined

[6] `http://www-sop.inria.fr/oasis/ProActive/`.
[7] Direct Acyclic Graph, see Sect. 2.2.5.

and scatter operations [57]), in the hypothesis that the set of resources is known in advance.

Structural performance models [59] are the first effort to develop compositional performance models for component applications. As discussed in Sect. 2.1.3, most scientific and Grid component models rely on the concept of the 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 [60, 61] 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 [60] propose parametric cost models for PIPE, FARM and MULTIBLOCK skeletons, that can be arbitrarily composed and nested. In [61], analytic cost models for applications, composed by PIPEs and DEALs, are derived within a stochastic process algebra formulation.

Trace-based performance models [62, 63, 64] 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 these applications, it is possible, with a certain degree of precision, to forecast the performance of the same or similar applications on different resources.

The problem of deriving a performance model for components has also been addressed in the context of component frameworks such as COM+/.NET [66], EJB [65] and CCA [67]. Such works apply to the analytical performance model (LQN) or trace-based performance model in order to derive a model for components. In [68], trace-based models are exploited to select the most suitable components out of multiple available choices for building an optimal application, from the point of view of performance.

2.1.5 Component Contracts

Recalling the definition of component in Sec. 2.1, it is clear that the whole lifecycle of a component is related to *contractually specified interfaces*. This concept is not new: in object-oriented design, the concept of design-by-contract [69] introduces the *object contract* (specified as pre- and post-conditions). When making components contract-aware [70], contracts can be divided in the following levels: *syntactic* (signatures of data types), *behavioral* (semantic descriptions of data types), *synchronization* (management of concurrency issues) and *Quality of Service* (management of all non-functional requirements and guarantees).

The essential elements needed to deal with these contracts are mechanisms to specify them, to verify their correctness and to enforce and monitor them at runtime. There have been many proposals for contract specification languages, both for syntactical/behavioral contracts (e.g. Jass [71], OCL [72], AsmL [73]) and for QoS contracts (QML [74], CQML [75], CQML$^+$ [76]). Every component is described in terms of quality attributes and the compositional/behavioral correctness is checked via matching/ordering relationships.

Commercial component architectures (.NET, CCM, EJB) describe components simply in terms of syntactical/behavioral contracts [77, 78]. Some scientific/Grid component frameworks introduce the concept of a performance contract [63, 79], while other works report studies on the forecast of the performance in distributed computing environments

[80]. In this research, the performance contract is defined as the forecast of performance of an application on a given computational resource. More precisely, given a set of resources which have certain capabilities and an application with given characteristics, a performance contract states the achievement of specified and desired performance. These contracts are used at runtime to monitor the performance of the application, to identify contract violations (which subsequently trigger corrective actions).

2.1.6 Discussion

Compositional component frameworks represent a promising solution to the problem of Grid applications development. Components seem well suited to efficiently face the new challenges, in terms of programmability, interoperability, code reuse, and efficiency, that mainly derive from the other features of Grids. Coupled with structured parallel programming methodologies, they might fill the gap between high performance computing and the Grid.

But to reach the concept of *invisible Grid* [36] several research challenges must be solved. A generic performance model for components must be identified and studied. In this thesis a performance model for a hierarchical component model is proposed. In the following paragraphs, the main differences between the proposed model and the existing one are highlighted.

Analytical performance models typically translate a parallel application into an analytic representation of its execution behavior and the target runtime system. This translation is usually not straightforward, it may require approximations to obtain mathematical models [81] 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 [53]. More complex models can be solved by means of simulation, but at the cost of a larger computation time.

Symbolic performance models share several properties with the proposed model: both can be extracted from the structure of programs, are parametric, and can be efficiently evaluated. However the main difference is that the presented model computes not only a lower bound, but the asymptotic steady-state performance of an application, that is in general a better approximation of the real performance.

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 will most likely 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. building a concrete model of components/applications to be exploited in their mapping on previously-unknown target platforms.

Structural performance models are extended by the presented model by proposing a methodology well-suited for the generic composition of skeletons and by taking into account the synchronization problems introduced by using streamed communications.

Trace information is exploited in the presented model, but this is done 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 is mapped on actual resources, historical information will be used to model the runtime behavior of single components. This information will then be coupled with the component interactions information to obtain a prediction of the performance of the whole application.

Analytical, symbolic and structural performance models need the full knowledge of the target platform to derive performance measures. Therefore, to compare the results of different mappings, they must be evaluated multiple times. The presented approach will decouple the modeling of the application performance from the target platform, allowing to evaluate the model once in order to derive enough information to drive the mapping process. Trace-based approaches are used to overcome such limitation, but they are not compositional. Therefore, they must be applied from scratch to every new application, even if it is built from a same set of components.

A hierarchical component model allows large scale Grid-aware applications to be built. As soon as the application requests to exploit a user-defined QoS, this approach can rapidly lead to intolerable complexity. The deployment and execution frameworks of applications of such complexity have to provide automated mechanisms to manage all low-level operations needed to enforce the desired QoS of the application. This can be obtained with the specification of a performance contract. The current approaches which exploits performance contracts are focused on the use of such contracts to verify at design time or to monitor at runtime the expected behavior of components. The approach to performance contracts proposed in this work completes such approaches introducing quantitative performance methodologies and measures exploitable both at launch time and runtime. A methodology to derive performance contracts for component-based application is proposed. Contracts must include the application performance model and its parameters, as well as the desired overall performance goal. To support adaptive components management, a contract may include runtime policies for (some of the) components. Policies specify how to react to dynamic changes, steering the application performance toward the goal. Exploiting such a contract we should be able to map the application components on Grid resources, and to generate Service Level Agreements (SLAs) with those resources, both at launch time, and at runtime, with suitable renegotiation of the SLAs.

2.2 Grid Computing

In 1996, Ian Foster and Carl Kesselman proposed the first version of a software toolkit *"to enable the construction of networked virtual supercomputers, or metacomputers, execution environments in which high-speed networks are used to connect supercomputers, databases, scientific instruments, and advanced display devices, perhaps located at geographically distributed sites"* [6]. Together with Steve Tuecke, they evolved the Globus Toolkit, incorporating not just CPU management but also storage management, security provisioning, data movement, monitoring and a toolkit for developing additional services based on the same infrastructure including agreement negotiation, notification mechanisms, trigger services and information aggregation. They incorporated a new set of previously ignored problems in the metacomputing and distributed computing

paradigms, leading to the very first definition of (computational) Grid [82]: *"A computational Grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities"*. The term Grid was chosen as an analogy to a power Grid that provides consistent, pervasive, dependable transparent access to electricity, irrespective of its source [5].

Considerable progress has since been made on the construction of such an infrastructure, and now the term Grid computing has grown far beyond its original intent. Three stages of Grid evolution can be identified [7]. The objective of the first generation systems was to provide computational resources to a range of high performance applications [83, 84]. They tried to overcome a number of issues to be able to work efficiently and effectively, including communications, resource management and manipulation of remote data. The second generation systems correspond to the vision of Grid computing given in [82]. New issues and old ones had to be confronted, including heterogeneity, scalability and adaptability. This second wave of Grid computing technologies relied on two success concepts to address such issues: middleware and (open) standards. *Middleware* is a set of services needed to support a common set of applications in a distributed network environment [85], and generally it is viewed as a software layer between operating systems and user applications, which provides a variety of services required by applications to work correctly. *Standards* are publicly available documents that contain implementable specifications, allowing increased compatibility between software entities compatible with the same specification. Middleware has been used to hide the heterogeneous nature of Grid resources and standards are the key factor which allows interoperability of different systems through common protocols, APIs[8] and services. Representative middleware systems of this second generation include the Globus Toolkit [86], Unicore [87] and Legion [88]. The third generation systems reflect the current status of Grid technologies. These systems involve increasing adoption of a service oriented model, which allows the reuse of existing software components, and also an increasing attention to metadata, to shift the view from the resources to applications. Moreover, with the increase of scale and heterogeneity, humans can no longer directly control the steering and execution of applications on Grid. This new issue forces the introduction of autonomy features at various levels of the Grid.

According to the Grid foundation paper [2] *"the real and specific problem that underlies the Grid concept is coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations"*. According to this statement, to give a clear definition of what a Grid is, a clear understanding of what a *virtual organization* represents is necessary.

> A virtual organization (VO) is a set of individuals and/or institutions who share access to computers, softwares, data and other resources for solving collaboratively problems in science, industry and engineering, in an highly controlled manner, with resource providers and consumers defining clearly and carefully just what is shared, who is allowed to share and the conditions under which sharing occurs.

[8] Application Programming Interfaces.

Given this definition of a VO, in [89, 90] a Grid is defined by the following list of minimum properties that must be exhibited by a system.

A grid is a system that:
1. Coordinates resources that are not subject to centralized control
2. Uses standard, open, general-purpose protocols and interfaces
3. Delivers nontrivial qualities of services

Note that this is not the definition of "the Grid" but the definition of "a Grid". In fact, there can be different types of Grids, depending on the virtual organizations they are addressing. The nature of the resources shared, the standards used and the definition of the ultimate qualities of service depend on the problems and the applications a virtual organization must solve/use.

2.2.1 Grid Resource Management

Generally, *Grid resource management* is the process of identifying application requirements, locating various types of capability, arranging for their use, utilizing them and monitoring their state [91] in order to run Grid applications as efficiently as possible[9]. A Grid Resource Management System (RMS) is central to the operation of a Grid, and its design must consider aspects including site autonomy, heterogeneity, extensibility, coallocation, scheduling, online control.

Although several Grid RMS have been proposed, currently there are none that supports the full set of functionalities required by a Grid RMS. This is due to the inherent complexity of the entire resource management process.

Current Grid systems are application-specific, either in research or commercial environments. Most of them are focused on the scheduling mechanisms to select when and where an application should be run. The Grid scheduler strongly reflects the organization of the resources boing part of a Grid. Three common Grid scheduling scenarios have been identified, according to the currently deployed Grid systems [92].

Enterprise Grids

Enterprise Grids represent a scenario of commercial interest in which the available IT resources within a company are better exploited and the administrative overhead is lowered by the employment of Grid technologies. The resources are typically not owned by different providers and are therefore not part of different administrative domains. In this scenario there is a centralized scheduling architecture; i.e. a central Grid scheduler is the single access point to the whole infrastructure and manages the resource manager interfaces that interact directly with the local resource managers (see Fig. 2.1).

High Performance Computing Grids

High Performance Computing (HPC) Grids represent a scenario in which different computing sites, e.g. scientific research labs, collaborate for joint research purposes. Here,

[9] This process has been defined as *Grid scheduling*, *meta-scheduling* and *resource brokering*.

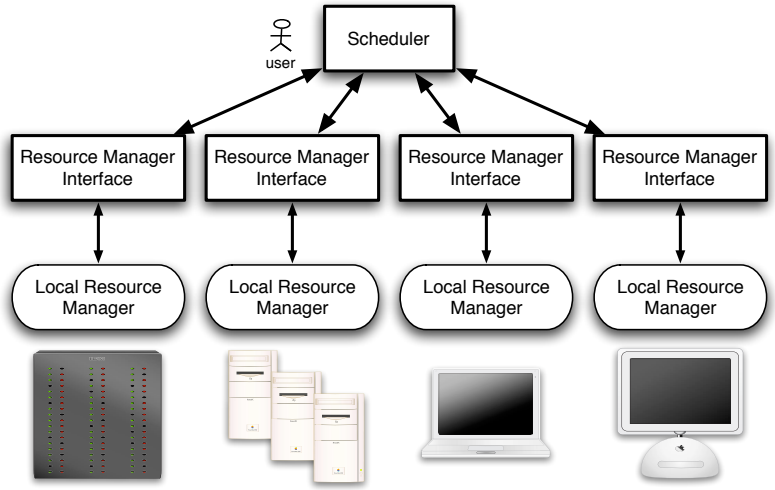

Fig. 2.1. Example of a scheduling infrastructure for Enterprise Grids.

compute- and/or data-intensive applications are executed on the participating HPC computing resources which are usually large parallel computers or cluster systems. In this case the resources are part of several administrative domains, with their own policies and rules. A user can submit jobs to a Grid scheduler at institute or VO level (see Fig. 2.2).

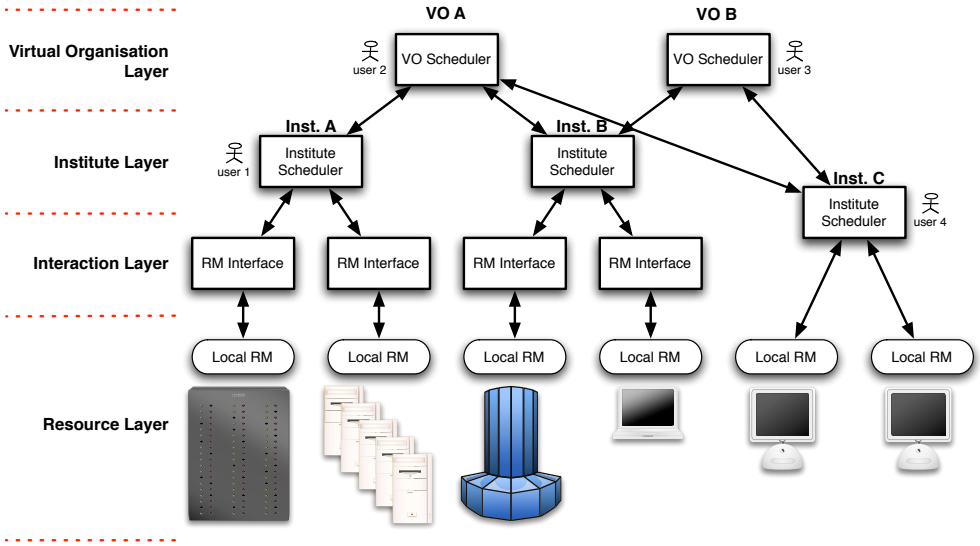

Fig. 2.2. Example of a scheduling infrastructure for HPC Grids.

Global Grids

Global Grids comprise very heterogeneous resources, from single desktop machines to large-scale HPC machines, which are connected through a global network. This scenario is the most general one. Introducing physical constraints on resources, their interconnections and their interaction we can fall back in the previous case. In this a fully decentralized architecture, any scheduler can accept jobs to be scheduled, forward them to other schedulers and fulfill scheduling requests on the behalf of any other scheduler (see Fig. 2.3).

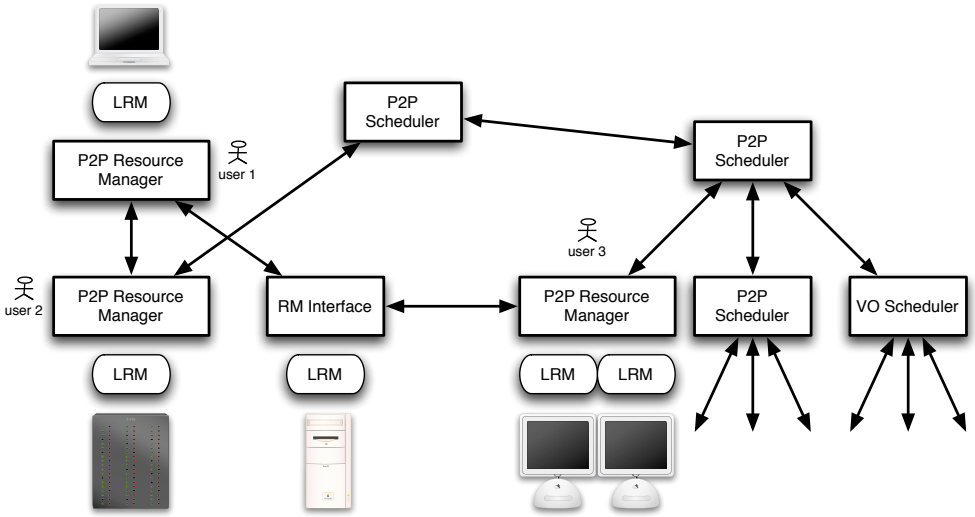

Fig. 2.3. Example of a scheduling infrastructure for Global Grids.

2.2.2 Grid Scheduling Process

The three scenarios illustrated in the previous section show several entities interacting to perform scheduling. To solve scheduling problems, these entities have to execute several tasks [93], often interacting with other services. A general architecture of the Grid scheduling process has been delineated in [94], identifying three major phases in the scheduling process: resource discovery and filtering according to application requirements, resource selection and scheduling, according to certain objectives, and job submission and monitoring (see Fig. 2.4).

Phase 1. Resource Discovery: This phase requires first a security mechanism to determine the set of resources that the user submitting the application has access to. Then a standard way to express application structure and requirements is necessary. Eventually, the list of accessible resources must be refined, excluding the resources that does not match the application requirements. Typically in this phase *static* resource requirements are considered.

Phase 2. System Selection: This phase determines the (best) set of resources that will be chosen to execute the application. This selection requires the gathering of detailed

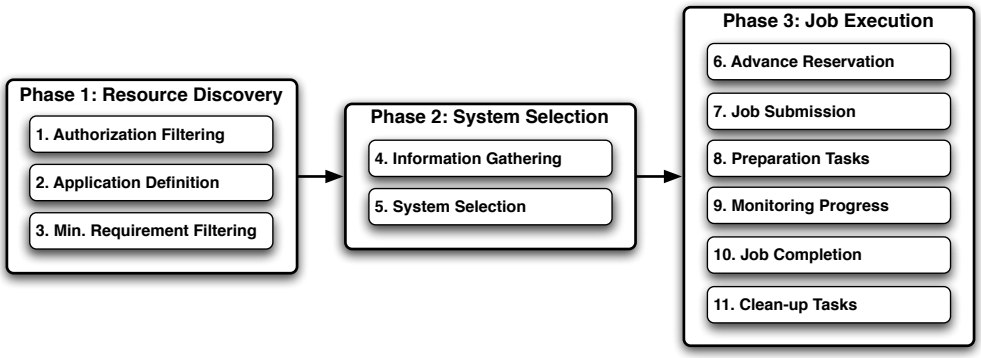

Fig. 2.4. Three phases process for Grid resource management

dynamic information from resources. This information is used to rank the remaining resources in the list and to select the best ones according to specific criteria. In the case of complex applications, a performance model may be required during this phase.

Phase 3. Job Execution. The last phase involves all the steps required in running a job. Once resources are chosen, the application can be submitted to the resources. Depending on the resources, part or all of them may have to be reserved in advance. Before the actual execution of a job, setup, staging, claiming a reservation, or other actions needed to prepare the resource to run the application. When the job is finished, the user is notified and temporary files/settings removed.

2.2.3 Grid Scheduling Algorithms

In [95], a hierarchical taxonomy for scheduling algorithms in parallel and distributed computing systems is proposed. This taxonomy has been reviewed in [96], focusing on Grid computing as a special distributed system. Grid scheduling uses information about the whole system to allocate tasks to multiple resources in order to optimize a system-wide performance objective. That is, Grid scheduling falls under the *global scheduling* category, in contrast to *local scheduling* algorithms, managing the allocation of a single resource. Then global scheduling algorithms are grouped in *static* and *dynamic scheduling algorithms*, based on the time the scheduling decisions are made. Both static [97, 98] and dynamic scheduling [99] are widely adopted in Grid computing. Besides other classifications, both families can be categorized further in *optimal* and *sub-optimal* algorithms. Optimal algorithms are able to find the minimum of an objective function, while sub-optimal ones do not find the optimum, but an "good" solution, according to some specified criteria. Another important characterization of scheduling algorithms concerns the nature of the input problem. The jobs/applications submitted to the scheduler can be completely specified *off-line* or their characteristics (or part of them) may be known only at submission time (i.e. *online*).

It should be clear that Grid scheduling covers a huge set of problem types. Each problem type can be characterized by different objective functions, applications characteristics, platform models, etc. Most of such problems are known to be NP-hard, so the procedure to determine an optimal solution can be exponentially complex. It is clear that

sub-optimal solutions are required to find "approximate" good solutions. Such solutions are calculated by algorithms that can be divided in the last two categories: *approximate* and *heuristics algorithms*. While the approximate algorithm aims to find sub-optimal solutions with tight bounds on their "goodness" (i.e. the distance between the optimal value of the objective function and the actual one), the heuristics algorithm cannot typically give optimality bounds on the solutions they found, but they "perform well" in a large category of the test cases. Most part of Grid scheduling solutions fall in this category.

2.2.4 Objective Functions

From the point of view of the objective functions, scheduling for Grids can be divided in two approaches: *resource-centric* and *application-oriented* [100]. The resource-centric approach focuses on the optimization of the performance of a resource or the whole Grid system, e.g. throughput, average response, or utilization. The application-oriented approach focuses on the optimization of the performance of individual applications/jobs. As well-known examples, AppLeS addresses specifically the scheduling aspects on an application level [101], while the Condor approach focuses on the optimization of idle cycles of CPUs working at resource level [102].

As the Service Oriented Architecture [103] embraces current Grid technologies [104], the *quality of services* is becoming central in many Grid applications. The meaning of QoS typically varies according to the goals of different users, but in general it represents a set of conditions (*service level objectives*, SLOs) grouped in an agreement (*service level agreement*, SLA) that must be respected to successfully execute an application [105]. Of course, the application runtime support must interact with the Grid infrastructure (through middleware and standards) to manage heterogeneity and dynamicity of the involved resources. Nevertheless, most current QoS concerns are at the resource management level rather than at the application scheduling level.

2.2.5 Application Models

There exists a large body of literature covering scheduling on heterogeneous platforms such as Grids. Such works differ not only on the Grid infrastructure model and the objective function to optimize, but also on the nature of the applications to schedule. The applications can be categorized according to different features: for example, we can have applications requiring a single resource (e.g. a single executable file), a set of homogeneous resources (e.g. a parallel application requiring a set of identical CPUs) or any kind of resources (e.g. a complex multidisciplinary application). The characteristics describing an application with respect to the objective function (e.g. deadline, reservation slots, execution time) or the execution environment (e.g. coallocation, security, licenses) can vary. Typically more information means higher complexity of the application and its scheduling algorithms.

A main dichotomy regards the dependencies between the tasks of an application. These kinds of dependencies arise when the execution of a set of tasks composing an application must be coordinated in order to correctly execute the application. Typical examples of task dependency are coallocation and precedence relationships. For independent task scheduling several heuristics have been proposed [97, 98, 106]. For dependent tasks,

two major problems are under investigation in the Grid community, namely workflow scheduling and coallocation. The former arises when the tasks composing an application have precedence order. In this case a popular model for the application is the Directed Acyclic Graph (DAG) [107], in which a node represents a single task and a directed edge represents the precedence order between its endpoint. In the case of coallocation, there is not a standard model adopted in Grid scheduling. In its base form, coallocation simply requires that the tasks of an application should be scheduled on several resources at the same time, and no additional information is provided to drive such allocation. In order to fulfill specific QoS requirements, more information about the structure and the behavior of the applications is required. In these cases, a promising model is represented by the Task Interaction Graph (TIG) [107], in which a node represents a single task and an undirected edge represent a communication between its endpoints.

2.2.6 TIG Scheduling

The TIG model and the related scheduling problem was introduced first in [108]. Here, a TIG was used to model sequentially executing tasks and the objective was to minimize the sum of computation and communication costs on a homogeneous platform assuming non-overlapping communications and computations [109]. As a result of the sequential nature of execution, in the case of identical processors, it will always be optimal to execute tasks on a single processor [110]. An optimal solution exists for the two processors case and in other particular cases (chain and tree TIGs) but the general case with four processors or more is known to be NP-complete [111].

Apart from the applications structure, the platform model and the objective function, the existing heuristics to address the scheduling of generic TIGs have been categorized according to the method used to explore the solutions space [112, 113]. The solution methods are categorized in the following methods: graph-theoretic, mathematical programming, state-space search, probabilistic/randomized optimization methods. In particular, a class of graph-theoretic methods use task clustering methods [114, 115]

2.2.7 Discussion

Grid computing represents both a great opportunity and a grand challenge. The vision of a *"globally interconnected set of computers through which everyone could quickly access data and programs from any site"* [10] [116] and the forecast that *"we will probably see the spread of 'computer utilities', which, like present electric and telephone utilities, will service individual homes and offices across the country"* [11][117] is becoming effective with the widespread adoption of Grid technologies. In addition to low-level tools required to uniformly access and exploit a huge set of heterogeneous resources, security concerns, application performance and efficient use of such resources are some of the main challenges that must be addressed by researchers to allow the pervasive use of such potential.

[10] Licklider, 1962, first head of the computer research program at DARPA, which leaded in 1969 to the creation of DARPANET, the world's first operational packet switching network.

[11] Kleinrock, 1969, creator of the basic principles of packet switching, the technology underpinning the Internet.

This work investigates the problem of resource management, focusing on the execution of component-based applications. At first glance, such applications are basically a set of executable files that must run concurrently to perform their functions, thus coallocation is a fundamental requirement. This is a current issue in Grid computing, and several prototypes and solutions have been presented in the past. With the new strong accent on Service Oriented Architecture (SOA), new improvements to coallocation should be brought in order to manage QoS requirements. Due to the dynamic nature of the Grids and virtual organizations, it is feasible to assume that there will not be a long-standing, high-level service infrastructure to execute component applications. Of course, the basic services needed by the middleware will be always available, but the runtime environments for such applications should be deployed and/or executed on demand depending on the application submission time and the status of the available Grid resources. Moreover, to enforce the user performance requirements, a more in-depth description of this type of application should be provided to steer the scheduling of the application components on the Grid.

In this thesis, the assumption of a hierarchical component model with clear communication semantics enables us to provide this information to new Grid resource management services. Exploiting well-known information providers for Grid resources, resource managers for execution environments [118] and new management services for network resources [119], this thesis investigates two new heuristics to exploit his information in the coallocation of components trying to find a scheduling fulfilling as close as possible the user contract.

3

Component Performance Model

A "suitable" description of a software component is needed to design and implement automatic deployment and execution tools. In particular, if a scheduler needs a description of an application we have to provide "numbers" (e.g. some type of weighted graph) about its structure and performance.

The following chapter illustrates a mathematical characterization of the behavior of a component. The main target of this characterization is the decoupling of the *structural* behavior from the *runtime* behavior.

The content of this chapter has been presented in [29, 120, 121, 122].

3.1 Component Structure

In general, a *composite component* is defined as a composition of components, obtained by connecting their interfaces. The building blocks of composite components are called *primitive components*, representing sequential as well as parallel computations whose structure (or behavior) is described with non-component technologies. In a hierarchical component model, an application can be seen as a *composite component* with no external interfaces. Communications between components are usually expressed through remote procedure calls or streams.

A composite component can be structured as a hypergraph whose nodes represent primitive components and whose (hyper)edges represent communications or synchronizations between components. Nodes (either representing primitive and composite components) 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.

3.1.1 Communications

In this thesis data-flow stream communications are studied. Communications between components are implemented through input/output interface bindings. Every primitive 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:

- **unicast**: one-to-one connection. Every item sent on the output stream interface is received in order by the input stream interface.
- **merge**: many-to-one connection. Every item sent on the output stream interfaces and 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.
- **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.

3.1.2 Computations

Primitive 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:

- **data parallel**: a single function is executed in parallel on different portions of the same data;
- **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 computations and delivers result items to its output streams.

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

A computation can be activated if the following conditions hold:

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

3.2 Dynamic Model

In this section the runtime behaviors of primitive components and streams are modeled, exploiting techniques borrowed from system theory [123]. In the following section this model will be used to derive a simpler description of a component behavior based on steady-state analysis.

3.2.1 Node Behavior

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

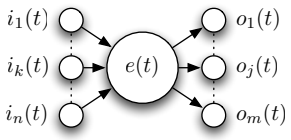

Fig. 3.1. Sequential component at runtime

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

- $i_k(t)$: total number of received items at time t from the k^{th} input interface;
- $e(t)$: total number of computations carried out at time t;
- $o_j(t)$: total number of sent items at time t through the j^{th} 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:

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

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:

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

where τ_{1k} represents the maximum number of elements that can be received on the k^{th} 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:

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

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 ($e_1(t), \ldots, e_s(t)$).

3.2.2 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 when a full item is produced:

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

The previous equation holds even if there is a sensible latency Δ in the transmission:

$$i(t) \leq \lfloor o(t - \Delta) \rfloor \leq \lfloor o(t) \rfloor \tag{3.6}$$

The edge implementation will exploit finite communication buffers and the network layer transfers chunks of data. Let q^{-1} be the minimum fraction of the item transferred atomically. This means that for each item to be transferred, actually there are q chunks of that item transferred physically through the network. Then

$$o(t) - \frac{\lfloor q \cdot i(t) \rfloor}{q} \leq \tau_2 \tag{3.7}$$

where τ_2 represents the maximum number of items that can be buffered. This equation embodies also the fact that an item is available at destination only when all of its chunks have been completely transferred. Therefore the maximum admissible value for $o(t)$ at time t is:

$$o^{max}(t) = \tau_2 + \frac{\lfloor q \cdot i(t) \rfloor}{q} \tag{3.8}$$

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:

$$o^{max}(t) - f\big(e_1(t), \ldots, e_m(t)\big) \geq 0 \tag{3.9}$$

For *merge* streams with k source interfaces and *broadcast* streams with k destination interfaces, the general constraints (Eqs. (3.5) and (3.7) for the unicast stream) become:

$$\text{merge: } \begin{cases} i(t) \leq \sum_k o_k(t) \\ \sum_k o_k(t) - i(t) \leq \tau_{2k} \end{cases} \tag{3.10}$$

$$\text{broadcast: } \begin{cases} \forall k \quad i_k(t) \leq o(t) \\ \forall k \quad o(t) - i_k(t) \leq \tau_{2k} \end{cases} \tag{3.11}$$

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

3.2.3 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: $i_{1,\ldots,n_i}(t), e_{1,\ldots,n_e}(t), o_{1,\ldots,n_o}(t)$. 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 this 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 MB/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* $u(x)$ (its value is 0 if $x \leq 0$, 1 otherwise), the number of performed (partial) computations per time unit is:

$$\frac{de}{dt} = u\Big(\min\Big(\lfloor i_1(t) \rfloor, \ldots, \lfloor i_n(t) \rfloor \Big) - e(t) \Big) \cdot u\Big(o^{max}(t) - f\big(e_1(t), \ldots, e_m(t)\big) \Big) \cdot \frac{w(t)}{l} \tag{3.12}$$

This equation simply states formally that a new computation is performed if there are new input items to be processed (first factor), i.e. the input buffers are not empty, and the output buffer is not full (second factor).

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

$$\frac{di}{dt} = u\Big(\lfloor o(t) \rfloor - i(t) \Big) \cdot u\Big(i^{max}(t) - i(t) \Big) \cdot \frac{b(t)}{s} \tag{3.13a}$$

$$\frac{di}{dt} = u\Big(\sum_k \lfloor o_k(t) \rfloor - i(t) \Big) \cdot u\Big(i^{max}(t) - i(t) \Big) \cdot \frac{b(t)}{s} \tag{3.13b}$$

$$\frac{di_k}{dt} = u\Big(\lfloor o(t) \rfloor - i_k(t) \Big) \cdot u\Big(i^{max}(t) - i_k(t) \Big) \cdot \frac{b(t)}{s} \tag{3.13c}$$

These equations state that a new transmission is performed if there are new output items to be sent on a stream (first factor) and the receiving input buffers are not full.

Note that an important assumption has been made. The work required to perform a computation is assumed to be *independent* from the values of the incoming items; their values are used only to perform computations[1]. 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:

$$\dot{p}(t) = U(p(t))\, \alpha(t) \tag{3.14}$$

We denoted with $U : \mathbb{P} \to \mathbf{M}_{n,n}$ the function that, for each 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 $b(t)$).

[1] This is called *ergodicity* in stochastic models.

We observe that the control matrix is piece-wise constant over non-infinitesimal time intervals; it derives from quantization in the general equations for the nodes (3.12), and in the equations for the streams (3.13). 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.

3.2.4 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:

$$\bar{p} = \mathbb{E}[\dot{p}|_{[t_0,\infty)}] = \int_{t_0}^{\infty} \dot{p}(t) dt = \lim_{t \to \infty} \frac{p(t) - p(t_0)}{t - t_0} \tag{3.15}$$

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 modeled associating each computation $e_k(t)$ with its activation rate

$$\bar{e}_k = \lim_{t \to \infty} \frac{e_k(t) - e_k(t_0)}{t - t_0} \tag{3.16}$$

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 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 be an index relative to an input stream activating computation e_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 summand 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 summand 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 2. *If the transfer function is (asymptotically) linear*

$$o_k = f_k(e_1, \ldots, e_m) = \alpha_k^1 e_1 + \ldots \alpha_k^m e_m + c_k(e_1, \ldots, e_m) \qquad \textit{with} \qquad \lim_{\|e\| \to \infty} \frac{\|c_k(e)\|}{\|e\|} = 0$$

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

$$\bar{o}_k = \sum_{i=1}^{m} \alpha_{ki} \bar{e}_i \tag{3.17}$$

Proof.

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

□

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

Proposition 3. *The steady-state transmission rate at the endpoints of a stream are characterized by the following balance equations:*

$$\textit{unicast: } \bar{o} = \bar{\imath} \tag{3.18a}$$

$$\textit{merge: } \bar{o}_1 + \bar{o}_2 = \bar{\imath} \tag{3.18b}$$

$$\textit{broadcast: } \bar{o} = \bar{\imath}_1 = \bar{\imath}_2 \tag{3.18c}$$

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

Proof. The proof is similar to the one of Prop. 1, exploiting:

- (3.5) and (3.7) for unicast,
- (3.10) for merge,
- (3.11) for broadcast.

□

3.2.5 Validation

The following example demonstrates that the presented model accurately models the dynamic behavior of a real application, and therefore can be exploited to derive further models more amenable to automatic manipulation.

The example is the video rendering pipeline presented in Fig. 3.2.

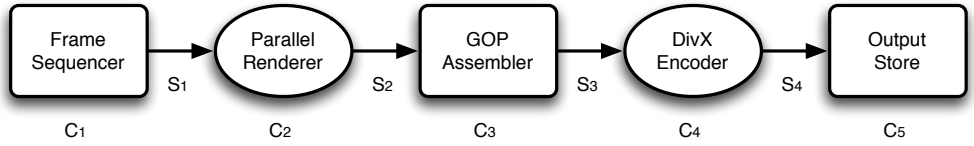

Fig. 3.2. Graph of the render-encode application

The first stage requests the rendering of a sequence of scenes while the second stage 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, and builds Groups Of Pictures (GOP), that are sent to the fourth stage, which performs DivX compression. The last stage collects DivX compressed pieces and stores them in an AVI output file.

The execution of the application has been conducted 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.

Note that each scene may require different computational time according to its complexity. To replicate this condition in the simulation, the sequential rendering time of each scene has been recorded. This information has been exploited to derive the dynamic parameters of the resources (computation-related components of $\alpha(t)$ in (3.14)). Information statically derived from the application code and execution platform has been exploited to determine the remaining parameters of the simulation (data sizes, buffer dimensions, network bandwidths, quantization constant).

Figure 3.3 shows a comparison between the evolution of the computation of the actual execution and the simulation performed using the analytical model. For each GOP, the absolute completion time measured from the start of the execution is displayed. The analytical model is able to accurately reproduce the dynamic behavior of the state variables of the application.

Figure 3.4 shows that the application behavior actually tends to steady-state.

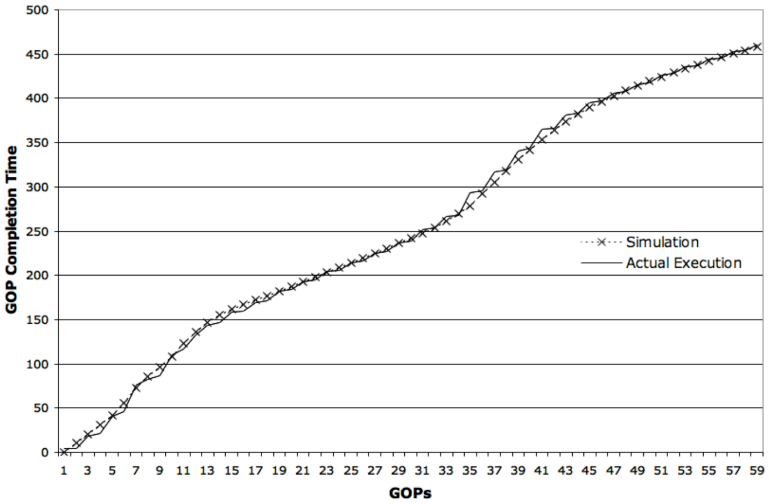

Fig. 3.3. Comparison of actual execution and simulation of the parallel renderer

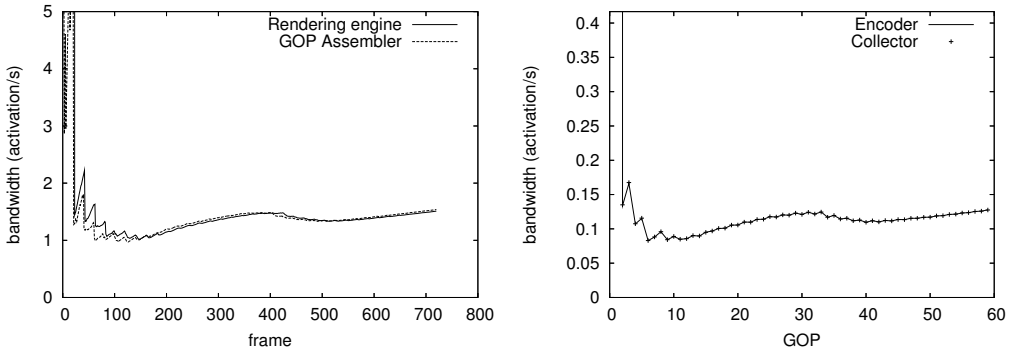

Fig. 3.4. Convergence to steady-state of averaged performance features

Performance features are measured as in (3.16), i.e. averaging the number of performed tasks on the duration of the computation. The left diagram shows the performance of the Render and the GOP Assembler nodes, which operate on frames. The right 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. 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).

Moreover, Fig. 3.4 shows that the averaged computation rates stabilize during the computation, allowing to adopt a steady-state model to approximate the actual application run.

3.3 Performance Model

Propositions 1, 2 and 3 can be operatively exploited to model the steady-state behavior of a program/component. The example in Fig. 3.5 shows how the steady-state behavior of a

generic component application is modeled within the proposed framework. An *execution*

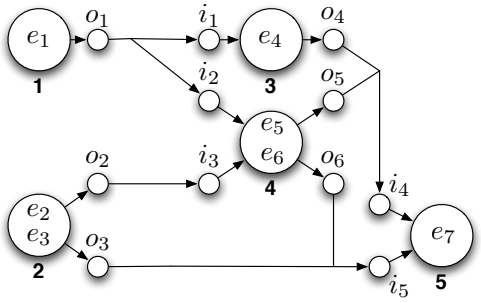

Fig. 3.5. Component-based application with two sources of data

rate e_k is associated to each computation (nodes 1, 3 and 5 represent sequential components, nodes 2 and 4 represent parallel components). To each input/output interface its *data transfer rate* (i_k and o_k respectively) is associated. These variables completely specify the application state from the point of view of its performance, therefore they will be called **performance features** of the application.

Prop. 1 allows to express which input interfaces activate each computation, directly relating the respective rates. If, for example, each computation in node 4 has its own input interfaces, then:

$$\begin{cases} i_1 = e_4 & \text{first input} \\ i_2 = e_5 & \text{second input} \\ i_3 = e_6 & \text{third input} \\ i_4 = e_7 & \text{fourth input} \\ i_5 = e_7 & \text{fifth input} \end{cases}$$

Note that the computation of node 7 is activated when there is a new item both on input 4 and input 5. In case of activation with every single new item, there should be a merging of input streams 4 and 5 in a single input interface in node 7.

Prop. 2 allows to express output rates as linear combinations of execution rates, provided that the related coefficients are known:

$$\begin{cases} o_1 = f_1(e_1) = a_1 e_1 & \text{node 1} \\ o_2 = f_2(e_2, e_3) = a_{22} e_2 + a_{23} e_3 & \text{node 2} \\ o_3 = f_3(e_2, e_3) = a_{32} e_2 + a_{33} e_3 & \text{node 2} \\ o_4 = f_4(e_4) = a_4 e_4 & \text{node 3} \\ o_5 = f_5(e_5, e_6) = a_{55} e_5 + a_{56} e_6 & \text{node 4} \\ o_6 = f_6(e_5, e_6) = a_{65} e_5 + a_{66} e_6 & \text{node 4} \end{cases}$$

These coefficients must be provided by developers of programs/components by means of some **component annotations** to the compiler.

Eventually, Prop. 3 allows to relate output rates to input rates on the four streams:

$$\begin{cases} i_1 = o_1 & \text{broadcast} \\ i_2 = o_1 & \text{broadcast} \\ i_3 = o_2 & \text{unicast} \\ i_4 = o_4 + o_5 & \text{merge} \\ i_5 = o_3 + o_6 & \text{merge} \end{cases}$$

The derived equations define the **performance model** of the example application; with this approach, a compiler exploiting simple annotations (the values of coefficients a_{ij} and the graph structure) can automatically find an analytical performance model for complex graph structures, obtaining the same predictive power provided by the performance models for skeletons, but it is not limited to few, well-known skeletons.

In general, denoting with **i** the vector of the input rates, with **e** the vector of the execution rates and with **o** the vector of the output rates, the component annotations can be expressed as systems of linear equations:

$$\mathbf{i} = A\mathbf{e} \tag{3.19}$$

$$\mathbf{o} = B\mathbf{e} \tag{3.20}$$

$$\mathbf{i} = C\mathbf{o} \tag{3.21}$$

where (3.19) relates the input rates to execution rates, (3.20) relates the execution rates to output rates and(3.21) describes the connection structure of the graph.

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**:

$$\left[\begin{array}{c|c|c} -I & A & 0 \\ \hline 0 & B & -I \\ \hline -I & 0 & C \end{array}\right] \left[\begin{array}{c} \mathbf{i} \\ \mathbf{e} \\ \mathbf{o} \end{array}\right] = \left[\begin{array}{c} 0 \\ 0 \\ 0 \end{array}\right] \tag{3.22}$$

The set of solutions of the system is a vector subspace of the space $\mathbb{R}^n$ (where n is the total number of variables, either input rates, output rates or execution rates); the dimension of the solution space gives the number of **degrees of freedom** of the application. If this dimension is 1, then the system is completely determined when 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 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 every assignment of positive values for the vector $\left[\begin{array}{ccc}\mathbf{i} & \mathbf{e} & \mathbf{o}\end{array}\right]^T \in \mathbb{R}^n$ that is a solution of the system is a possible "operation point" for the modeled application.

This can be mathematically stated by the side conditions:

$$\left[\begin{array}{c} \mathbf{i} \\ \mathbf{e} \\ \mathbf{o} \end{array}\right] \geq \left[\begin{array}{c} 0 \\ 0 \\ 0 \end{array}\right] \tag{3.23}$$

The solution space, then, is a convex, possibly infinite polyhedron.

The approach outlined before is efficient, in fact the simplification of the simultaneous equations can be achieved using well known techniques from linear algebra, for which a fast algorithms has been designed (see Chapter 4).

3.4 Composite Components

The presented model is suitable to describe single components (and therefore whole programs, as a special case), starting from their fine grain structure as graph of components.

Now consider the composite component with two input and two output interfaces shown in Fig. 3.6.

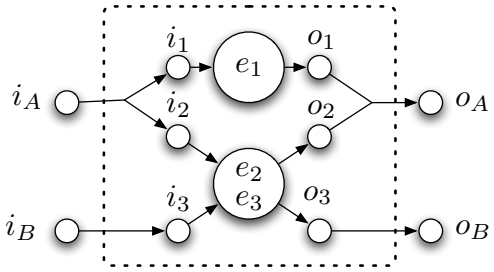

Fig. 3.6. Composite component

When designing the component, the internal structure is known, so the full model can be constructed:

$$\begin{cases} i_A = i_1 = i_2 & \text{input interface } i_A \\ i_B = i_3 & \text{input interface } i_B \\ i_1 = e_1 & \text{activation } e_1 \\ i_2 = e_2 & \text{activation } e_2 \\ i_3 = e_3 & \text{activation } e_3 \\ o_1 = a_1 e_1 & \text{computation } o_1 \\ o_2 = a_{22} e_2 + a_{23} e_3 & \text{computation } o_2 \\ o_3 = a_{32} e_2 + a_{33} e_3 & \text{computation } o_3 \\ o_A = o_1 + o_2 & \text{output interface } o_A \\ o_B = o_3 & \text{output interface } o_B \end{cases}$$

In the equations of the component model some variables appear, that can be identified with the interfaces of the composite component, and others that are related to its internal composition; moreover, the equations describe exactly the connection between its constituent components. To describe only the observable behavior of the component (i.e. the relations between its input and output interfaces) the observable variables (in this case i_A, i_B, o_A and o_B) must be discerned from the internal variables.

To do so, write the system of simultaneous equations in matrix form as $Ax = 0$, ordering the vector $x = [i_1, i_2, i_3, e_1, e_2, e_3, o_1, o_2, o_3, i_A, i_B, o_A, o_B]^T$ (and therefore the columns of A) with the internal variables before the observable ones.

$$\begin{bmatrix}
-1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & a_1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & a_{22} & a_{23} & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & a_{32} & a_{33} & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1
\end{bmatrix}
\begin{array}{c}
i_A = i_1 \\
i_A = i_2 \\
i_B = i_3 \\
i_1 = e_1 \\
i_2 = e_2 \\
i_3 = e_3 \\
o_1 = a_1 e_1 \\
o_2 = a_{22} e_2 + a_{23} e_3 \\
o_3 = a_{32} e_2 + a_{33} e_3 \\
o_A = o_1 + o_2 \\
o_B = o_3
\end{array}$$

The Gauss-Jordan elimination algorithm simplifies systems of simultaneous equations producing an block upper triangular system. In other words, as in the example, the last two equations will relate only the observable variables. The algorithm performs several steps, each one consisting in the elimination of one variable (this gives the name to the algorithm) from all the equations below the one considered. In its basic form, the algorithm can perform row exchange (swapping equations), but not column exchange to the matrix (this would imply a reordering of the components of the solution vector x). The elimination is achieved, conceptually, by solving the considered equation for that variable, and substituting the solution in the following equations.

Note that the chosen ordering for x, in which internal variables appear before observable ones, is intended to produce equations in which only the observable variables appear, because the internal ones have been substituted by equivalent expressions involving only the observable ones.

The resulting matrix is:

$$\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & a_1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & a_{22} & a_{23} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & a_{32} & a_{33} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -a_1 - a_{22} & -a_{23} & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -a_{32} & -a_{33} & 0 & -1
\end{bmatrix}$$

The resulting system can be split in two parts:

1. a first part relating internal quantities to the external ones (the first nine equations)
2. a second part relating only observable quantities (the last two equations).

The second part can be seen as the black-box model of the composite component, and can be used to enforce the encapsulation of the internal structure of each component.

The first part, instead, is useful when, once solved the high-level performance model of the program to fulfill the performance constraints, it is necessary to propagate them to primitive components, in order to drive the mapping mechanisms as well as the dynamic adaptation mechanisms.

3.5 Comparison with Queueing Network Theory

The results of the presented model for the steady-state behavior of a component resembles the job flow analysis performed when studying queuing networks. This section discusses the similarities and the differences between the two approaches.

A component is described by a set of equations, which describe the flow of tasks in the application: in the previous example, every task produced by a stage is sent over the stream to the next stage. More complex situations can be accommodated: when multiple destinations are possible for tasks departing from a node, each destination has an associated coefficient, that can be seen as the probability that a task will follow a specific route, exactly as in the queuing network framework. Moreover, the ergodicity of the queueing network is the hypothesis commonly assumed to be able to mathematically solve the network, The ergodicity condition implies the hypothesis of Prop. 2, i.e. the transfer function is asymptotically linear, with routing probabilities as coefficients.

The proposed model extends the possible set of behaviors for the nodes, in fact a computation in a node, activated by a task, can produce more than a single task as output, or, as well, can absorb a number of input tasks to produce a single output. These behaviors are captured by the same coefficients, which now are seen as the product of the routing probability by the task multiplication/division factor.

A known limitation of queuing network models is that they cannot model event synchronization. The proposed model overcomes this limitation for the recurring, unconditional synchronizations happening when a single computation is activated by the simultaneous presence of a token on every associated input interface. To demonstrate this, the following example (see Fig. 3.7) shows that the proposed model captures deadlocks occurring in malformed programs.

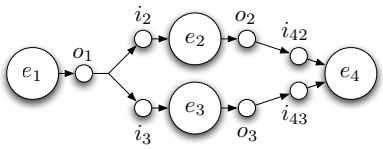

Fig. 3.7. A program that deadlocks if i_{42} and i_{43} are synchronized to activate the computation e_4

The following equations describe the example in Fig. 3.7:

$$\begin{cases} o_1 = e_1 & \text{comp. 1 output} \\ o_1 = i_2 = i_3 & \text{broadcast} \\ i_2 = e_2 & \text{comp. 2 input} \\ o_2 = 3e_2 & \text{comp. 2 output} \\ i_3 = e_3 & \text{comp. 3 input} \\ o_3 = 2e_3 & \text{comp. 3 output} \\ o_2 = i_{42} & \text{unicast} \\ o_3 = i_{43} & \text{unicast} \\ i_{42} = i_{43} = e_4 & \text{comp. 4 input D} \end{cases}$$

If the simultaneous equations are solved, the result is that the system has zero degrees of freedom, and all the rates must be zero at steady-state. This means that every infinite

computation would eventually stall. This does not imply that cannot exist finite computations that complete successfully; indeed, a trivial computation of a zero-length stream is an example of successful computation. However, this shows that the limited form of synchronization of the computational model is captured by the performance models.

4

Component Performance Contract

Given a component-based application, the user will typically require a minimum performance level during the execution of the application. Component applications can easily grow in size, and the normal user could not be aware of the whole structure of the application. In composing an application, a user can exploit components provided "as-is" without a complete knowledge of their implementation. Nevertheless such components can be structured as composite components on their own.

Such a user will therefore require a limited set of performance, for example the minimum frame rate for a visualization of the results of a computation or the minimum number of record operations per minute on a database collecting experimental results.

Such performance requirements can be easily translated in requirements on execution, input or output rates of some components of the whole application. The application programming framework/runtime support is in charge of deriving the necessary conditions to be respected in order to guarantee such performance.

Section 4.1 shows a formal model of performance requirements, and a detailed analysis of the impact of such requirements on the application model presented in Chap. 3 is shown. Section 4.2 propose an algorithm to derive the performance requirements on each component of an application given a small set of requirements on the whole application. Section 4.3 shows how to related the performance requirements with the runtime platform. Eventually, Sect. 4.4 propose a definition of performance contract, according to the definitions of performance features, annotations, model and requirements.

The content of this chapter has been presented in [121, 122, 124].

4.1 Performance Constraints and Requirements

Given the definitions of performance features and the performance model given in Chap. 3, a **performance constraint** is defined as an inequality in the form:

$$x_i \geq c_i \tag{4.1}$$

where x_i is a performance feature (input, output or execution rate) and c_i is a strictly positive real number representing the desired operational value of the feature x_i. A constraint is *satisfied (at runtime)* if the measured value of the associated performance feature (x_i) is greater than or equal to the specified value (c_i).

Let a set of k performance constraints be given by the user. Such user-provided constraints are called **performance requirements**. They represent the user expected QoS of some application components at runtime. When this happens, the following question arises: "What values should the other performance features assume at runtime to satisfy the performance requirements?". We are looking for is a methodology to derive performance constraints on each feature of the performance model, given the interactions between the components and few performance requirements.

Let the vector $x \in \mathbb{R}^n$ indicate the performance features; the vector $x_c \in \mathbb{R}^k$ represents the features constrained by the user requirements, and $x_{uc} \in \mathbb{R}^{n-k}$ indicates the unconstrained features. For simplicity, the elements of x are ordered in such a way that $x = [x_{uc}|x_c]^T$. Moreover, $c \in \mathbb{R}^k$ denotes the vector of the performance requirement values.

In order to express every performance feature as a function of the constrained features, the rows of the performance model matrix are reordered in such a way that the constrained variables appear last (according to the ordering of x). Then exploiting the Gauss-Jordan elimination algorithm, the performance matrix is row reduced into block-echelon form.

The application graph in Fig. 4.1 presents an example of this.

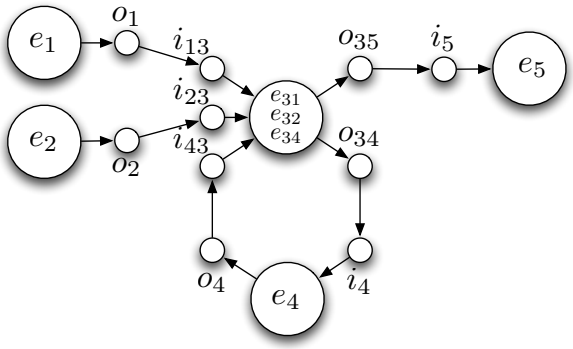

Fig. 4.1. Example application graph for equations reordering

The performance features characterizing the performance model are:

$$\mathbf{i} = \begin{bmatrix} i_{13} \\ i_{23} \\ i_{43} \\ i_4 \\ i_5 \end{bmatrix} \quad \mathbf{e} = \begin{bmatrix} e_1 \\ e_2 \\ e_{31} \\ e_{32} \\ e_{34} \\ e_4 \\ e_5 \end{bmatrix} \quad \mathbf{o} = \begin{bmatrix} o_1 \\ o_2 \\ o_{34} \\ o_{35} \\ o_4 \end{bmatrix}$$

Each input interface activates the corresponding computation in nodes 4 and 5. Computations in nodes 1 and 2 are spontaneous. The first computation of node 3 is activated by data from node 1, the second one is activated by data from node 2 and the third one by data from node 4. The equations relating output interfaces to node computations

must be provided by developers, in order to fully specify the performance model. The equations reported here are an example. Therefore, the structure of the application is captured by the following equations:

1. **Input interfaces**

$$\left.\begin{array}{l} i_{13} = e_{31} \\ i_{23} = e_{32} \\ i_{43} = e_{34} \\ i_4 = e_4 \\ i_5 = e_5 \end{array}\right\} \Rightarrow \mathbf{i} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{e}$$

2. **Output interfaces**

$$\left.\begin{array}{l} o_1 = e_1 \\ o_2 = e_2 \\ o_{34} = \frac{3}{4}\, e_{31} + \frac{4}{5}\, e_{32} + \frac{9}{10}\, e_{34} \\ o_{35} = \frac{1}{4}\, e_{31} + \frac{1}{5}\, e_{32} + \frac{1}{10}\, e_{34} \\ o_4 = e_4 \end{array}\right\} \Rightarrow \mathbf{o} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{3}{4} & \frac{4}{5} & \frac{9}{10} & 0 & 0 \\ 0 & 0 & \frac{1}{4} & \frac{1}{5} & \frac{1}{10} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \mathbf{e}$$

3. **Communications**

$$\left.\begin{array}{l} i_{13} = o_1 \\ i_{23} = o_2 \\ i_{43} = o_4 \\ i_4 = o_{34} \\ i_5 = o_{35} \end{array}\right\} \Rightarrow \mathbf{i} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \mathbf{o}$$

Next, the performance matrix is calculated. Note that for this example the system has been partially reduced to improve readability by eliminating some variables without loss of generality. From an algorithmic point of view, the whole system can be analyzed in the same manner. The analysis is limited to the variables e_{31}, e_{32}, e_{34}, e_4 and e_5[1]. Reducing the previous system the following equations are obtained:

$$\begin{cases} e_1 = e_{31} \\ e_2 = e_{32} \\ e_4 = e_{34} \\ e_4 = \frac{3}{4}\, e_{31} + \frac{4}{5}\, e_{32} + \frac{9}{10}\, e_{34} \\ e_5 = \frac{1}{4}\, e_{31} + \frac{1}{5}\, e_{34} + \frac{1}{10}\, e_{34} \end{cases}$$

Substituting o_{34} and o_{35} with i_4 and i_5 first, then with e_4 and e_5 respectively, the following homogeneous system is obtained:

$$\begin{bmatrix} \frac{3}{4} & \frac{4}{5} & \frac{9}{10} & -1 & 0 \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{10} & 0 & -1 \\ 0 & 0 & 1 & -1 & 0 \end{bmatrix} \begin{bmatrix} e_{31} \\ e_{32} \\ e_{34} \\ e_4 \\ e_5 \end{bmatrix} = 0$$

[1] The unicast streams and the omitted transfer functions do not introduce new degrees of freedom.

At this point, depending on the degrees of freedom of the performance model and the type of the performance requirements provided by the user, it may be possible to derive the constraints on the unconstrained features.

In the following examples, different choices of the constrained features are selected, and the matrix is subsequently transformed with the Gauss-Jordan algorithm.

1. e_{34} and e_4 are constrained:

$$\left[\begin{array}{ccc|cc} \frac{3}{4} & 0 & -12 & \frac{3}{2} & 3 \\ 0 & -\frac{1}{5} & -3 & -\frac{3}{5} & 1 \\ 0 & 0 & 0 & 1 & -1 \end{array}\right] \left[\begin{array}{c} e_{31} \\ e_{32} \\ e_5 \\ e_{34} \\ e_4 \end{array}\right] \Rightarrow \left\{\begin{array}{l} e_{31} = 16e_5 - 2e_{34} - 4e_4 \\ e_{32} = -15e_5 - 3e_{34} + 5e_4 \\ e_{34} - e_4 = 0 \end{array}\right.$$

   In this case, the constraints can not fully specify the system. Yet, the last row does express the correlation between e_{34} and e_4 which must be respected. It is impossible, however, to specify e_5 (the variable related to the third column) using only the constrained variables. In this case the performance model is said to be **under specified** by the set of user constraints.
2. e_{31} and e_5 are constrained:

$$\left[\begin{array}{ccc|cc} \frac{4}{5} & 0 & 0 & \frac{4}{5} & -\frac{4}{5} \\ 0 & -\frac{1}{2} & 0 & -\frac{1}{4} & 4 \\ 0 & 0 & 1 & \frac{1}{2} & -8 \end{array}\right] \left[\begin{array}{c} e_{32} \\ e_{34} \\ e_4 \\ e_{31} \\ e_5 \end{array}\right] \Rightarrow \left\{\begin{array}{l} e_{32} = -e_{31} + e_5 \\ e_{34} = -0.5e_{31} + 8e_5 \\ e_4 = -0.5e_{31} + 8e_5 \end{array}\right.$$

   In this case each unconstrained variable can be expressed as a linear combination of the constrained variables. Here, the performance model is **fully specified** by the set of constraints. Nevertheless it is also possible to obtain negative solutions (e.g. choosing $e_{31} = 2$ and $e_5 = 1$), which imply that the constraints can not be fulfilled.
3. e_{31}, e_{32} and e_5 are constrained:

$$\left[\begin{array}{cc|ccc} \frac{9}{10} & 0 & \frac{9}{4} & \frac{9}{5} & -9 \\ 0 & \frac{1}{9} & \frac{1}{6} & \frac{1}{9} & -1 \\ \hline 0 & 0 & -1 & -1 & 1 \end{array}\right] \left[\begin{array}{c} e_{34} \\ e_4 \\ e_{31} \\ e_{32} \\ e_5 \end{array}\right] \Rightarrow \left\{\begin{array}{l} e_{34} = -2.5e_{31} - 2e_{32} + 10e_5 \\ e_4 = -1.5e_{31} - e_{32} + e_5 \\ \hline e_5 + e_{31} - e_{32} = 0 \end{array}\right.$$

   In this case the unconstrained variables can be expressed as a linear combination of the constrained variables, but there is a relation between the constrained variables expressed by the last row, and the constraints must fulfill it. In this case, the performance model is **over specified** by the set of constraints.

4.2 Constraints Resolution Algorithm

The form that the performance matrix assumes as a result the Gauss-Jordan transformation can be used to recognize whether the set of constraints is well specified, if new constraints should be added or if some constraints should be relaxed.

In general a performance model with n performance features and m equations is fully specified by a set of k constraints if the result matrix has the form

$$\left[\begin{array}{c|c} I_{(n-k)\times(n-k)} & D_{(n-k)\times k} \\ \hline 0_{(m-n+k)\times(n-k)} & E_{(m-n+k)\times k} \end{array}\right] \tag{4.2}$$

Note that $m - n + k$ can be zero, and in that case the form is

$$\left[\begin{array}{c|c} I_{m\times m} & D_{m\times(n-m)} \end{array}\right] \tag{4.3}$$

4.2.1 Fully Specified Models

When the system is fully specified, it is easy to solve the final system to find the minimum values of the application unconstrained variables. In fact, recalling that the vector x_{uc} (x_c) denotes the unconstrained (constrained) variables and the vector c the performance constraints values, by substituting x_c with c the following holds:

$$x_{uc} + Dc = 0 \qquad \text{that is equivalent to} \qquad x_{uc} = -Dc \tag{4.4}$$

If negative solutions are found, it is possible to compute the nearest set of constraints $c' = c + h$ with $h \in \mathbb{R}^k, h \geq 0$ such that $x_{uc} = -Dc' \geq 0$, solving the following linear optimization problem:

$$\begin{array}{l} \min \sum h_i \\ \text{s.t.} \\ \begin{cases} x_{uc} = -D(c+h) \\ x_{uc} \geq 0 \\ h \geq 0 \end{cases} \end{array} \tag{4.5}$$

In this way it is easy to find the correct values for the constrained variables $x_c = c + h$ and for the unconstrained variables x_{uc}. Note that the constraints $h \geq 0$ assure the new values of x_c respect the original constraints.

4.2.2 Over Specified Models

When the model is over specified, there are too many constraints with respect to the degrees of freedom of the model, and thus the values of the constrained features cannot be arbitrarily fixed. When this occurs, the performance matrix has the following form:

$$\left[\begin{array}{c|c} I & M_1 \\ \hline 0 & M_2 \end{array}\right] \tag{4.6}$$

where the matrices M_1 and M_2 have different sizes with respect to Eq. (4.2). For this to work, the following must be satisfied:

$$M_2 x_c = 0 \tag{4.7}$$

By substituting x_c with c in Eq. (4.7), the system is not verified for some c_i, and instead the following holds:

$$M_2 c = d \tag{4.8}$$

At this point, a new set of constraints $c' = c+h$ with $h \in \mathbb{R}^k, h \geq 0$ must be chosen, which satisfies $M_2 c' = 0$ and the need for the new values of the constraints to be as close as possible to the user specified ones. Moreover, the admissible values for the unconstrained variables are positive numbers and therefore $-M_1 c' \geq 0$ must also be satisfied.

To compute c' different techniques can be used. To do this, it is possible to simply minimize $\sum h_i$ subject to the previous constraints:

$$\begin{array}{l} \min \sum h_i \\ \text{s.t.} \\ \left\{ \begin{array}{l} M_2 h = -d \\ -M_1(c+h) \geq 0 \\ h \geq 0 \end{array} \right. \end{array} \tag{4.9}$$

and determine the new values $c' = c + h$ for the constrained variables and the values $x_{uc} = -M_1(c + h)$ for the unconstrained variables.

In the previous example it holds $x_{uc} = [e_{31}\ e_{32}\ e_5]^T$; if $c = [1\ 1\ 1]^T$ then the last relation is violated ($d = 1$). The minimization problem to solve is:

$$\begin{array}{l} \min h_{31} + h_{32} + h_5 \\ \text{s.t.} \\ \left\{ \begin{array}{l} h_{31} + h_{32} - h_5 = -1 \\ h_{31} \geq 0 \\ h_{32} \geq 0 \\ h_5 \geq 0 \end{array} \right. \end{array}$$

In this simple case, the solution of the problem can be found by direct inspection: $h_{31} = 0$, $h_{32} = 0$ and $h_5 = 1$.

4.2.3 Under Specified Models

The aim of the proposed approach is to uniquely identify the performance requirements on every component. An under specified model has an infinite number of solutions. The application of the simplex method to the under specified model would find a wrong solution, in the sense that it is not the one desired by the user.

For example, consider a simple application with two independent nodes that send data to a third node through a merge stream. Consider only the variables o_1, o_2 and i_3 that characterize the stream, and the structure equation $o_1 + o_2 = i_3$.

If a user specifies the performance constraint $i_3 \geq 5$, the equivalent linear optimization problem should be:

$$\begin{array}{l} \min o_1 + o_2 + (i_3 - 5) \\ \text{s.t.} \\ \left\{ \begin{array}{l} o_1 + o_2 - i_3 = 0 \\ o_1 \geq 0 \\ o_2 \geq 0 \\ i_3 \geq 5 \end{array} \right. \end{array}$$

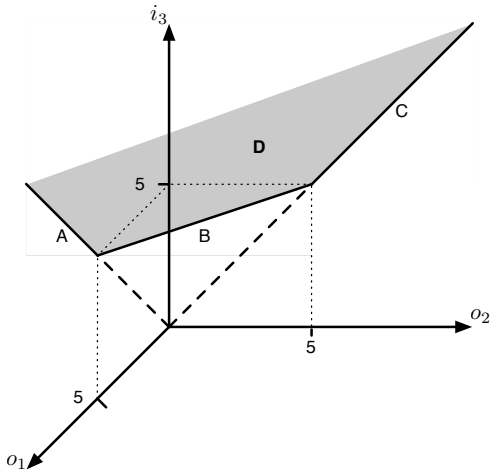

Fig. 4.2. Polyhedron of the LP problem in the example

A visual representation of the polyhedron of the example is depicted in Fig. 4.2, where the polyhedron D is the planar surface delimited by the lines A, B and C.

The gradient of the function to be minimized projected on this surface is normal to the line B; it means that every point on this line is a solution of the LP problem. However, the simplex algorithm will return a vertex (either $[5\ 0\ 5]^T$ or $[0\ 5\ 5]^T$), without providing any warning about the infinite number of solutions. The solutions found in this way hardly are the ones intended by the user (why he put two components producing the stream, if in the solution only one is working?). This suggests that such indeterminate conditions should be flagged as errors, so that the user can better detail the behavior he desires from the application.

The following algorithm summarizes the whole procedure.

Input: performance matrix M, requirements values vector c
Output: constraint values vector or an error message

```
1  M1 = column-reorder(M);
2  M2 = gauss-jordan(M1);
3  if (M2 is fully specified) then
4  |  return [c | −Dc];
5  else if (M2 is over specified) then
6  |  h = linear-program(M2, c);
7  |  return [c | −D(c + h)];
8  else
9  |  return error("Too few requirements");
10 end
```

Algorithm 1: The performance requirements propagation algorithm

4.2.4 Composite Components Constraints

So far, it has been shown how to derive the input, output and execution rates of a component, either primitive or composite, given its structure (the performance model

and its features) and a set of (few) performance requirements. Given such information, the methodology proposed in the previous sections can be used to derive the performance constraints on every feature that are as close as possible to the high-level performance requirements requested by the user.

Given a general multi-component application, it can be seen as a collection of interacting components. Such components may be primitive or composite. Clearly, to proceed to the mapping of the application, the performance constraints on every primitive component must be known, in order to identify the requirements for every computation and communication. Once the constraints on the application components are known, they can be projected to the composite components. The constraints obtained on the input/output interfaces of a composite component can be considered as performance requirements for its inner structure. At this point it is easy to apply the proposed methodology again to the graph of inner components. Note that, by construction of the model of composite components (as in (3.22)), the constraints will univocally determine the values of the relative sub-components, so just the reordering and the Gauss-Jordan triangulation steps are required.

This procedure may be recursive. In fact, composite components may have different composite components in their implementation, but this recursion will eventually end, because the final building blocks are always primitive components.

4.2.5 Complexity

The computational complexity of the algorithm is polynomial in time, w.r.t. the input size n of the performance features. The time complexity $T_1(n_0)$ of the algorithm is given by

$$T_1(n_0) = T_{GJ}(n_0) + T_{LP}(n_0) \tag{4.10}$$

where n_0 is the number of observable performance features (typically n_0 is greater than the number of constraints), $T_{GJ}(n_0)$ is the time complexity of the Gauss-Jordan algorithm ($O(n_0^3)$) and $T_{LP}(n_0)$ is the complexity of the Karmarkar algorithm for the linear optimization problem ($O(n_0^{3.5})$).

The time complexity $T_2(n_i)$ for the projection of the constraints in the generic i^{th} composite component is

$$T_2(n_i) = T_{GJ}(n_i) \tag{4.11}$$

where n_i is the number of the inner performance features of the i^{th} composite component.

Considering the input size n of the whole algorithm as the sum of the observable performance features and all the inner performance features, the time complexity $T(n)$ is given by:

$$\begin{aligned} T(n) &= T_1(n_0) + \sum_{i=1}^{k} T_2(n_i) = O(n_0^{3.5}) + O(n_0^3) + \sum_{i=1}^{k} O(n_i^3) \\ &= O(n_0^{3.5}) + \sum_{i=0}^{k} O(n_i^3) = O(n_0^{3.5}) + O(n^3) \end{aligned} \tag{4.12}$$

Typical values for n (which roughly correspond to the total number of component interfaces in the application) are in the range between few tens up to several hundreds, for

reasonably sized Grid applications. The algorithm run time is usually a small fraction of the total time needed to determine the application mapping and start the application. The longest parts consists of retrieving all candidate resources and staging the applications binaries onto remote machines.

4.3 Performance Annotations

So far, the proposed theory has only been concerned with the components performance, their relationships and the performance of a component-based application given some user requirements.

At this point, a major part of information needed so far does not need human intervention and can be obtained by automatic inspection on the composition of components. Now, the human intervention is needed to provide:

1. **component annotations** for each component, describing the steady-state relationships between computations and output interfaces (see Sect. 3.3) and
2. performance requirements, describing the user desired QoS for the whole application.

While the performance requirements are needed for every execution of the application (different users/runs can require different QoS), the component annotations should be given just once, by the component developer. Assuming a good level of expertise, the component developer must be very accurate in providing such information, and it can be attached to the component as a component *metadata*.

However, the mapping of the application components on a target platform, requires additional information. It is now necessary to state clearly the dependencies between component computations and computing resources and between stream communications and network resources.

Computations can be characterized in terms of the *work* they perform per activation (e.g. number of floating point operations, amount of data exchanged with memory or disk). The hypothesis assumed in Sect. 3.2.3 permits such activation parameters to be considered independent from the actual values of the received items.

Different metrics are possible (e.g. [125]) and multiple metrics can be considered simultaneously. Thus in general the work performed by a computation can be measured as an array of the values of the considered metrics, $\boldsymbol{l} = [l_1, \ldots, l_n]$. Such values can be obtained in two ways: by profiling or by historical records. Profiling can be done by the component developer or provider, while historical records can be obtained via controlled executions on reference resources. It is clear that these values depend on the characteristics of the computing resource. Therefore, one metric could be measured in MFlop/act (floating point operations required) and another in MB/act (megabytes of memory required).

Likewise, a computing resource can be described through work performed in the time unit, called subsequently *bandwidth*. Each computation metric has an associated resource bandwidth, e.g. a computational bandwidth (measured in MFlop/s) or a memory bandwidth or an I/O bandwidth (measured in MB/s). A computational resource should be described at minimum with the bandwidths that corresponds to the component work metrics.

With a single work metric l (e.g. MFlop/act), it is easy to translate a performance constraint c (e.g. act/s) into a resource bandwidth requirement w (MFlop/s) and vice-versa; for example:

$$w = l \cdot c \tag{4.13}$$

Once the number of activations per second c is known, its inverse represents the service time of an activation on the given resource.

With multiple metrics, given a work vector $\boldsymbol{l} = [l_1, \ldots, l_n]$ and a bandwidth vector $\boldsymbol{w} = [w_1, \ldots, w_n]$, Eq. 4.13 is still valid to derive the individual values, but the service time results:

$$t_s = \bigoplus_{i=1}^{n} \frac{l_i}{w_i} \tag{4.14}$$

where $\bigoplus$ is a combinator to be selected. For example, the combinator can be a sum operator if the work per activation is done sequentially, or a max operator for work carried out in parallel. In this case, the inverse of the service time represents the number of activations per second that the resource is able to provide with respect to the particular activation described by $\boldsymbol{l}$.

Eventually, communications are easily described simply by the size (in bytes) of the items flowing through the streams. In this way it is to check if a stream with a certain input rate (number of items flowing per second) and a certain item size can be "mapped" on a particular network resource characterized by its bandwidth (typically measured in Mb/s).

The characteristics discussed up to now should be provided as metadata for each component; again, the component developer, having the deepest knowledge of the component implementation, should be in charge of providing this information. This information is required to correctly map the entire application on a target environment with heterogeneous resources like a Grid, then they will be called **deployment annotations**. Deployment annotations are not restricted to this type of annotations, e.g. they may specify the minimum hardware requirements to run a component. This is important to note because some measures such as the work performed by a computation can be affected by a particular kind of resource. For example, a matrix multiplication operation, if it is executed on a machine with sufficient memory, requires a certain number of floating point operations and data fetch from main memory. Otherwise, the operating system will perform some I/O operations (page swapping) to overcome memory limitations, which will dramatically change the performance of the program; in this case, the execution time would be dominated by the I/O operations, that are orders of magnitude slower than memory access.

Placement constraints are provided as deployment annotations; in this way it is possible to specify that a particular component should be instantiated on a particular resource, or inside a given administrative domain (e.g. for privacy issues) or on a resource that has a copy of a given file or installed software. Both component and deployment annotations are called **performance annotations**.

4.4 Performance Contract

The information provided by performance annotations, supported by the values of the activation and input/output rates provided by the solved constraints, is sufficiently detailed and low level that it can be exploited by an automatic program launcher performing resource selection and mapping on behalf of the user. Therefore it s incorporated in the description of the application at launch time through a **performance contract**.

Now it is possible to define a performance contract for a component, primitive or composite as a list of metadata containing the following items

- the component performance model, provided by the component and/or application developers;
- the component performance requirements, provided by the end-user;
- for each primitive component, deployment annotations for its computations, provided by the component developer (through profiling) or automatic tools (through execution traces analysis);
- for each composite component:
  - a performance contract for the composite component;
  - a mapping of its external performance features to the ones of the inner subcomponents.

Clearly the performance contract for a component-based application is that of its topmost component. A performance contract is said to be **assessed** if the performance requirements have been propagated to each component and the constraints for every performance feature have been calculated

4.4.1 Validation

The following demonstrates the applicability of the proposed approach to select resources for a test application. The application is depicted in Fig. 4.3, also used in Sect. 3.2.5.

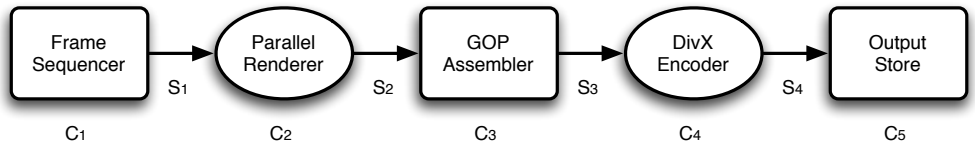

Fig. 4.3. Graph of the render-encode application

For groups of pictures of 12 pictures, the performance model for the application is :

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

and has one degree of freedom.

Suppose that the user wants 1 frame/sec at the last stage (the constraint is expressed by$C_{5i} \geq \frac{1}{12}$, because each input for C_5 is composed by 12 frames). By applying the performance model, the required computation and the transfer rates for each computation

and communication are derived. These values, paired with performance annotations (see Tab. 4.1) on the weight of computations or communications, 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, the requirement for stream $S_2 = C_{2o}$ is discussed. Since

Table 4.1. Deployment annotations for the 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	-

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

it is required to carry 1.19 MB messages with at least rate 1 message/s, a link of 9.52 Mb/s is sufficient. Likewise, the test application will never scale above 10 frames/s with a 100 Mb/s network. Such bandwidth can guarantee 10.5 messages per second. If a frame corresponds to a message, the application needs to be redesigned, if higher performances are required.

According to the discussion in Sec. 4.3, each computation is described by the vector $\boldsymbol{l} = [l_{MFlop}, l_{MB}]^T$, specifying the number of floating point operations and the data transferred to/from the main memory per activation. Resource bandwidths are described by the vector $\boldsymbol{w} = [w_{MFlop/s}, w_{MB/s}]^T$ and execution time is roughly estimated[2] as

$$t(\boldsymbol{l}, \boldsymbol{w}) = \frac{l_{MFlop}}{w_{MFlop/s}} + \frac{l_{MB}}{w_{MB/s}}$$

This model can be employed also to find an appropriate degree of parallelism for parallel computation nodes. In fact, it is possible to relate $t(\boldsymbol{l}, \boldsymbol{w})$ for an aggregate resource characterized by the bandwidths $\boldsymbol{w_1}, \ldots, \boldsymbol{w_n}$. Assuming perfect speedup:

$$t(\boldsymbol{l}, \boldsymbol{w}) = \left(\sum_{i=1}^{n} t(\boldsymbol{l}, \boldsymbol{w_i})^{-1} \right)^{-1}$$

In this way it is possible to derive, for each computation node, matching resource requirements. These will concern single resources for sequential nodes, and aggregate resources for parallel nodes.

In Fig. 4.4, two execution runs with different mappings (top: run on homogeneous clusters of Athlons XP 2600+, bottom: run on a set of heterogeneous resources: 9 P4@2GHz, 1 Athlon XP 2800+, 1 P4@2.8GHz) for the same constraint are displayed. At a first review it is notable that, even if the heterogeneous run has more variance in achieved bandwidth,

[2] We assume for simplicity no pipelining and latency hiding.

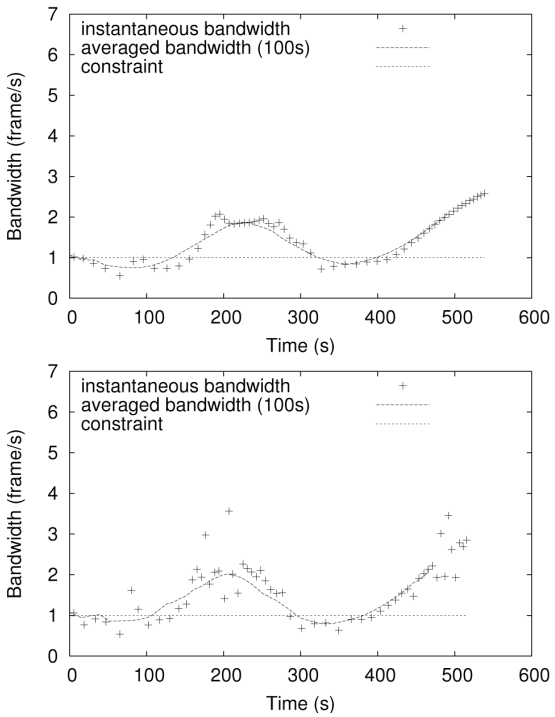

Fig. 4.4. Two executions of the application on homogeneous and heterogeneous resources.

the average bandwidth is comparable with the homogeneous one. This provides evidence that the employed performance model correctly handles heterogeneous sets of resources, for determining the correct parallelism degree (i.e. the number of replicas of C_2 and C_4 components). 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 schedule FPU bound computations (e.g. rendering) on machines with faster FPU.

The obtained results are as expected: the mapping computed using the performance model fulfills the constraint, during the begin of and throughout most of the application run time. This occurs because, in order to build the model, the achieved performance has been sampled on the first frames of the movie, but the application workload slightly changes with the evolution of the movie. This is unavoidable in complex applications, and may require dynamic rescheduling [13].

Although the heterogeneous run shows greater variance in the achieved bandwidth, the average bandwidth is comparable with the homogeneous case. This provides evidence that the performance model properly handles heterogeneous resources.

5

Component Applications Scheduling on Hierarchical Grids

This chapter proposes and discusses a launch-time scheduling heuristics which aims to schedule component-based parallel applications on *hierarchical Grids*. The goal of this heuristics is three-fold: to meet the minimal component computational requirements, to try to maximize the throughput between communicating components, and to evaluate on-the-fly the resource availability in order to minimize the effects of the dynamic changes of resource information (i.e. the aging effect). The evaluation of the proposed heuristics has been included, using simulations which apply it to a suite of task graphs and Grid platforms randomly generated, as well as further tests to schedule a real application on a real Grid.

The content of this chapter has been presented in [126, 127].

5.1 Introduction

The problem of component scheduling in a structured Grid environment deals with finding the proper assignment of components to Grid resources in order to optimize a performance metric. In a dynamic environment, this optimization goal is very complex. The general problem of scheduling a set of tasks on distributed resources is known to be very complex and even if some special scheduling problems are solvable in polynomial time, the dynamicity of the Grid can make an optimal or quasi-optimal solution useless in short times. Rather than elaborating very complex heuristics to try to optimally map components on a Grid, it is preferable to devise a launch-time scheduling heuristics which is able to find a schedule that can guarantee the QoS required by the end-user (a *feasible* schedule).

Current Grid resources are usually distributed in a clustered fashion. Resources in the same local network usually belong to the same organization and are relatively more homogeneous and less dynamic in a given period. Moreover, the hierarchical structure of the network, organized in local, metropolitan and wide are networks (LANs, MANs and WANs respectively) make easier the collection of coarse information on network performance. These characteristics can be exploited to steer the scheduling of a component-based application in such a way as to to allocate highly interacting components to the same local area network. This is done in order to minimize the use of unreliable network resources shared between a large number of users, i.e. in MANs and WANs.

5.2 Application Model

The application model presented in Sec. 3.3 is refined, considering the actual implementation of multicast and merge communication patterns. At resource level, merge and multicast are both implemented with distinct network links (see Fig. 5.1).

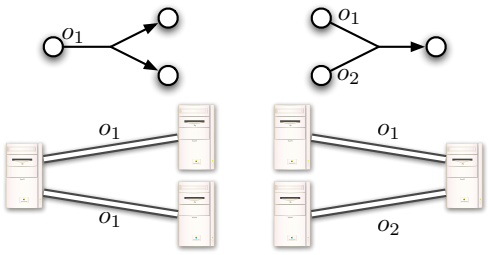

Fig. 5.1. Implementation of multicast (left) and merge (right) streams

In the case of multicast streams, every network link must provide a transmission bandwidth at least equal to the stream performance requirement (o_1 in Fig. 5.1), while for merge streams every network link must provide a different bandwidth, depending on the stream source (o_1 and o_2 in Fig. 5.1).

By transforming these types of streams (hyperedges in the original graph model) into simple links, the application graph can be seen as a simple graph. Besides this refinement, in the following the information on the directionality of the streams will not be exploited. In this way, the application graph $G_{App} = (N, V)$ can be viewed as a weighted Task Interaction Graph (TIG) [107]. In G_{App} a vertex $n_i \in N$ models a computation, and a undirected edge $a_{ij} \in V$, with $i \neq j$, models a communication occurring between vertex n_i and vertex n_j. Vertices have an associated weight w_i representing the minimum computational bandwidth to execute the computation respecting the associated performance requirement, and edges have an associated weight w_{ij} representing the amount of data exchanged between two vertices in the time unit. As discussed in Sec. 4.3, such weights can be obtained either by static code analysis or by executing the code on reference systems.

5.3 Platform Model

Three different kinds of networks are modeled: Wide Area Networks (WANs), Metropolitan Area Networks (MANs) and Local Area Networks (LANs). A *hierarchical graph* (i.e. a dendrogram [128]) denoted as $G_{Grid} = (LN, MN, WN)$ is adopted to model the Grid. LN is the set of nodes that represents sets of the Grid's computational resources (i.e. LANs), while MN and WN abstracts the MANs and WANs respectively. The root of the graph is a "virtual node" grouping together all the WANs. Every node in the dendrogram is a set of resources. The LAN set contains the resources to be exploited for tasks execution, while higher-level nodes contain all the resource of the lower-level nodes. Figure 5.2 shows a dendrogram including two MANs.

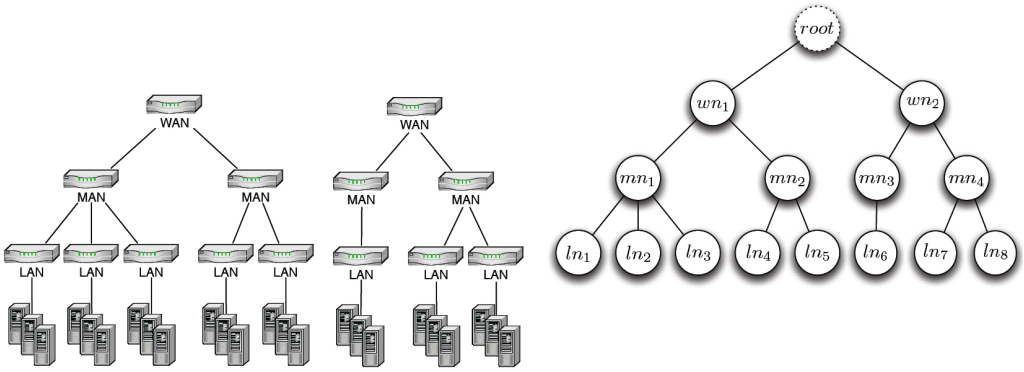

Fig. 5.2. Grid network topology (left) represented as a dendrogram (right)

Within the proposed approach, the basic assumption is that intra-LAN communications are characterized by higher bandwidth and a lower level of congestion with respect to any other network communication.

It is assumed that for the $j-th$ computational resource in LAN i $m_{ij} \in an_i$, its *nominal computational bandwidth* L_{ij} and its *percentage of current computational load* c_{ij}^{load} are known. The *current computational bandwidth* of a resource l_{ij} is computed as follows:

$$l_{ij} = L_{ij} \cdot (1 - c_{ij}^{load}) \tag{5.1}$$

It is assumed that each computational resource supports a multiprogramming environment. The *current* and *average computational bandwidth of a LAN* an_i containing k hosts are computed through the following formulas:

$$l_i = \sum_{j=1}^{k} l_{ij} \tag{5.2}$$

$$\overline{l_i} = \frac{l_i}{k} \tag{5.3}$$

The *current computational bandwidths* of a *MAN* and a *WAN* are defined in a similar way.

In terms of communication links, the *current bandwidth* $b(an_i, an_j)$ is assumed to be known, where an_i and an_j are two dendrogram vertices with the same father. In this way, only information about the bandwidths between LANs in the same MAN, MANs in the same WAN and between all the WANs is required. Both the structure of the dendrogram and the characteristics of the networks can be obtained using tools like *Network Weather Service* [129] and *TopoMon* [130].[1]

[1] The former returns the network and computational resource performances, while the latter returns an abstraction of the network topology.

5.4 Algorithm Architecture

The algorithm takes in input the G_{App} and G_{Grid} graphs, and it is structured according to two phases.

Phase 1 (Clustering): The goal of this preprocessing step is to elaborate G_{App} to find a set of subgraphs (called *clusters*) $S = (S_1, \ldots, S_k)$ by grouping the tasks with high inter-communication costs. The clustering is carried out by using the ϵ-approximation correlation clustering algorithm proposed in [131]. The goal of the clustering algorithm is to partition the application graph in clusters exploiting a similarity degree associated to each edge (i.e. trying to put nodes that have higher communication cost among them into the same subgraph). A positive similarity degree between two nodes means that the algorithm should try to put the two linked nodes in the same cluster, while a negative number will force the two nodes in two different clusters. Basically, the clustering problem is formulated as an integer linear programming problem with boolean variables. This problem is then relaxed and solved with standard LP methods and the solution is rounded to the integer solution exploiting the Region Growing technique [132]. This clustering algorithm does not need a meta-parameter (e.g. number of clusters), and the following quantity, derived from the edge weights, is considered as the similarity measure:

$$s_{ij} = \log\left(\frac{w_{ij}}{w^{max} - w_{ij}}\right) \tag{5.4}$$

where w^{max} is the maximum value of edge weights in the application graph. The more communication cost between two components (i.e. w_{ij} value), the more confidence that the tasks are similar. As a result, components within the same cluster need higher communication bandwidth than those belonging to different clusters. Therefore, according to the heuristics assumption that the better communication quality is inside a LAN with respect to MANs and WANs, the algorithm attempts to allocate all the tasks within a cluster onto the machines within the same LAN.

Phase 2 (Scheduling): The components within the clusters have to be scheduled onto Grid resources. First, the clusters in S are arranged in descending order with respect to their minimal computational bandwidth $w(S_i)$:

$$w(S_i) = \sum_{j \in S_i} w_j \tag{5.5}$$

Then, adopting a *priority-based* policy, the clusters are scheduled onto the resources of suitable LANs to run each cluster's tasks.

Selected a cluster and starting from the user LAN (ln_u), a *Closest First Search* (CFS) is applied to G_{Grid} in order to find a suitable LAN to host all the tasks within the selected cluster. The Closest First Search first visits the local LANs (i.e. the LANs in the user MAN) in descending order of current link bandwidth, measured from the user LAN. Then it moves upward to the user MAN, and again visits the MANs, in the user WAN, in decreasing order of current link bandwidth, measured from the user MAN. Finally, it moves upward visiting the WANs following the same bandwidth order. Note that, in general, the search does not visit all the nodes of the dendrogram, but it ends when enough resources are found to satisfy the application requirements. With a certain bias,

this heuristics tries to minimize the communication overhead, and therefore performance degradation by MAN and WAN network links, by putting strongly communicating nodes within single LANs.

The LAN ln_j *suitability* with respect to cluster S_i is computed according to the following formula:

$$\text{Suitability}(S_i, ln_j) = \frac{\overline{l_j[S_i]}}{\overline{w(S_i)} + \sigma_w(S_i)} \tag{5.6}$$

where $\overline{l_j[S_i]}$ is the mean computational power offered by the more powerful $min(|S_i|, |ln_j|)$ resources of the LAN ln_j[2], $\overline{w(S_i)}$ is the mean minimal computational power requested by the cluster:

$$\overline{w(S_i)} = \frac{w(S_i)}{|S_i|} \tag{5.7}$$

and $\sigma_w(S_i)$ is the standard deviation of the cluster computational power:

$$\sigma_w(S_i) = \sqrt{\frac{\sum_{k \in S_i} \left(w_k) - \overline{w(S_i)} \right)^2}{|S_i|}} \tag{5.8}$$

The LAN ln_j is suitable for S_i if and only if $\text{Suitability}(S_i, ln_j) \geq 1$. The tasks within a cluster are scheduled onto the machines of the first suitable LAN in CFS order.

The tasks are arranged in descending order with respect to their computational requirement (i.e. the w_i value), and then they are scheduled to the suitable LAN's machines by adopting a *priority-based* policy. The machine onto which schedule a task is selected according to the following parameter:

$$\text{Affinity}(n_i, m) = \frac{l_m}{w_i} \tag{5.9}$$

The resource m is suitable for a task n_i if and only if $\text{Affinity}(n_i, m) \geq 1$. Every task is scheduled on the machine with the highest affinity rank in the suitable LAN. The allocation of the task n_i on the machine m causes a reduction of l_m equals to w_i.

Since statistical information is used to select a suitable LAN, the chosen LAN may not be able to host all the tasks inside a cluster. It could happen that some tasks do not find any suitable machine. To overcome this problem, unallocated tasks are scheduled onto machines of a LAN as close as possible to the other tasks in terms of locality and available bandwidth in the network tree.

This process is repeated until all the cluster's tasks are allocated or until the root of the G_{Grid} graph is reached. In this last step the scheduling algorithm fails, ending without a valid allocation.

The following algorithms summarize the entire procedure.

[2] Note that if the number of tasks is greater than the number of machines, $\overline{l_j[S_i]}$ corresponds to the mean computational of the LAN, as in (5.2).

Input: Set of component clusters S, Grid topology G_{Grid}
Output: A list L containing feasible allocation of components to resources or an error message

```
1  Initialize the scheduling list L;
2  Initialize a pending nodes set P;
3  Order clusters s ∈ S into a cluster queue Q_C in decreasing order of w(s)
   values;
4  repeat
5      s = Q_C.removeFirst();
6      Get first LAN g in G_Grid such that Suitability(s, g) ≥ 1 by visiting
       G_Grid from the user LAN in a CFS fashion;
7      if (g ≠ ∅) then
8          P = LocalScheduling(s, g);
9      else
10         return error("Unable to find a feasible scheduling");
11     if (!P.empty()) then
12         NeighborSearch(P, g);
13 until (Q_C.empty()) ;
```

Algorithm 2: The hierarchical scheduling algorithm

Input: A set of component nodes N, a Grid node (LAN, MAN, WAN or root) g
Output: A set of pending nodes P

```
1  Initialize a pending node set P;
2  Order nodes n_i ∈ N into a component queue Q_N in decreasing order of w_i
   values;
3  repeat
4      n_i = Q_N.removeFirst();
5      Get resource m ∈ g with highest value of Affinity(n_i, m);
6      if (Affinity(n_i, m) ≥ 1) then
7          L.insert(n_i, m);
8          l_m = l_m − w_i;
9          if (l_m = 0) then G_Grid.remove(m);
10     else
11         P.add(n_i);
12 until (Q_N.empty()) ;
13 return P;
```

Algorithm 3: The local scheduling function `LocalScheduling`(N, g)

5.5 Performance Evaluation

This section shows an evaluation of the performance of the scheduling solution carried out by the proposed algorithm. The objective is to investigate the effect of the applications computational and communication requirements on reaching the algorithm goals. The evaluation has been conducted by simulations applying the algorithm to a suite of task

Input: A set of pending nodes P, a Grid node (LAN, MAN, WAN or root) g
Output: A set of pending nodes P^*

```
1 Initialize a pending node set P*;
2 Order g brothers in G_Grid in a Grid nodes queue Q_G in decreasing order of
  bandwidth values (measured from g);
3 repeat
4 |  g* = Q_G.removeFirst();
5 |  P*.addAll(LocalScheduling(P, g*));
6 until (P.empty() || Q_G.empty()) ;
7 if (P*.empty() ∧ g ≠ root) then
8 |  g* = father of g; NeighborSearch(P*, g*);
9 if (P*.empty()) then return error("Unable to find a feasible
  scheduling");
```

Algorithm 4: The neighbor search procedure `NeighborSearch(`P, g`)`

graphs and Grid platforms randomly generated. Moreover, a more thorough test was conducted by using a real life example of a rendering application.

5.5.1 Simulation Environment

To carry out a set of meaningful statistical information to evaluate the proposed heuristics a large number of simulations was conducted. As there is no existing information on parameterizing the algorithm, several parameter sets as first examples are assumed. Because of this, it cannot be concluded that these parameters are optimal nor feasible for other job scenarios. The evaluation exploits randomly generated TIGs with 8, 16, 32, 64 and 128 nodes with a variable number of edges. the weights of both the nodes and the edges weights are randomly generated from a uniform distribution $U(1, 10)$.

In order to evaluate the effect of the tasks' communication cost on the LAN configurations, TIGs were clustered changing the correlation function to obtain 1-3, 4-6 and 7-9 tasks per cluster for each generated TIG, i.e. the more the value of the correlation degree, the more is the communication similarity value and therefore greater is the number of tasks inside a cluster. To calculate the performance constraints, four different QoS level have been selected. Each QoS level specifies the performance constraints on the graph nodes with weight 1[3], and the QoS values are 25, 50, 75, 100 MFlop/s.

The Grids platforms have been generated to satisfy the following constraints:

1. Grids structured as real-world wide area grid by adopting specific topologies and realistic features for the computational machines and communication links;
2. The grid topology must be organized with a hierarchical structure (i.e. WANs, MANs and LANs);
3. The values of the bandwidth and the latency of each link must reflect those of real grid environments, especially by considering the different communication bandwidths among LAN, MAN and WAN networks.

[3] The performance constraints for the other nodes can be found multiplying the nodes weights by the provided QoS value.

The previous requirements are satisfied by the *Tiers* tool [133]. The machines' loads and links have been obtained by referring to the traces stored in the *Network Weather Service* web site, where information of several hosts connected in different networks are available. Grids were then generated with 32, 64, 128, 256 and 512 machines, grouped in LANs with 4, 8 and 16 machines. Then, a value representing the computational power has been associated to each host on the basis of a uniform distribution with values from 600 to 1300 MFlop/s. In order to run all the tests, 240 application TIGs and 15 Grids were generated.

5.5.2 Performance Metrics

To validate the results obtained a greedy *best fit* scheduling was implemented and it has been applied to every test case. To evaluate the schedules carried out by the proposed algorithm three different criteria are exploited: the percentage of scheduling failures, the component-resource affinity, the LAN hit ratio and the intra-cluster "distance" among components allocated on different Grid nodes.

The percentage of scheduling failures indicates the ratio of test applications which both algorithms failed to schedule. This can be due to the fact that it was impossible to find a schedule for the application or the selected algorithm failed to find one of the existing solutions. In the following criteria, the average values are computed discarding the simulations resulting in scheduling failures.

The component-resource affinity represents how many times the power of the resource on which a component is allocated is greater than the task minimal computation requirement. Hence, this parameter gives us an accurate estimation of the satisfaction level obtained by the algorithms in allocating a component.

The LAN hit ratio shows how many components of a cluster are allocated onto the resources of the suitable LAN. According to the algorithm hypothesis, the more the LAN hit ratio increases the more the throughput between communicating components increases.

Finally, the intra-cluster "distance" represents a scheduling distribution to estimate the reduction of the communication quality among the components of a cluster when some of them are not allocated in the suitable LAN.

These results have been obtained running the algorithms on a Pentium4 processor with a clock frequency of 3.0GHz and a memory of 1GB.

5.5.3 Evaluation for synthetic Grid scenarios

To evaluate the schedules carried out by the proposed algorithm, first the percentage of scheduling failures is evaluated. Figure 5.3 and Figure 5.4 show the percentage of scheduling failures with respect to the number of tasks per TIG and the number of tasks per cluster respectively. As expected increasing the number of tasks or the size of a cluster causes an exponential increment of the number of failures. In Figure 5.5 the algorithm is compared with the scheduling carried out by a greedy scheduling algorithm. This algorithm simply orders the task queue in decreasing order of node weights and the resource queue in decreasing order of computational power and it performs a *best fit*

scheduling. As can be seen, in the simulations, the percentage of failures of the greedy algorithm is lower than the percentage failure of the proposed one.

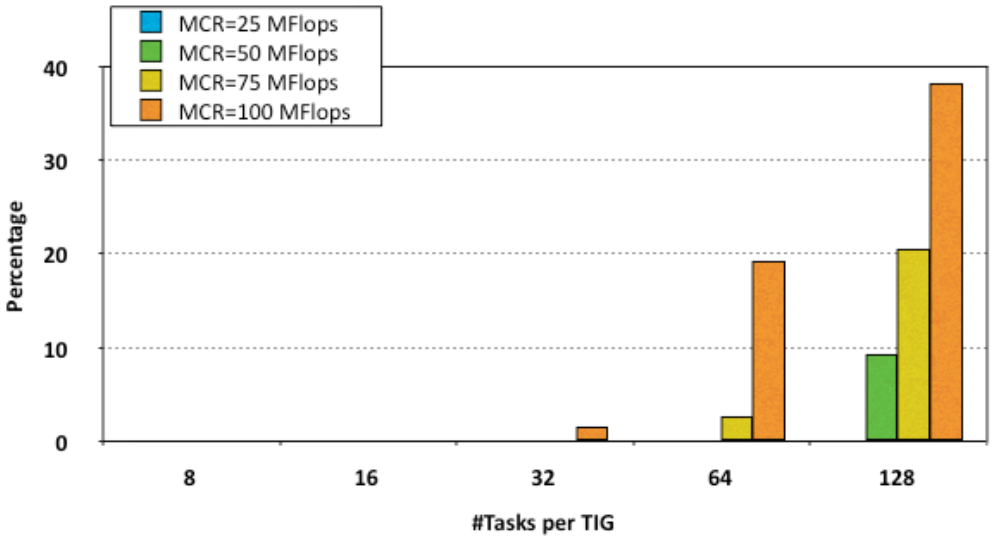

Fig. 5.3. Percentage of scheduling failures (1)

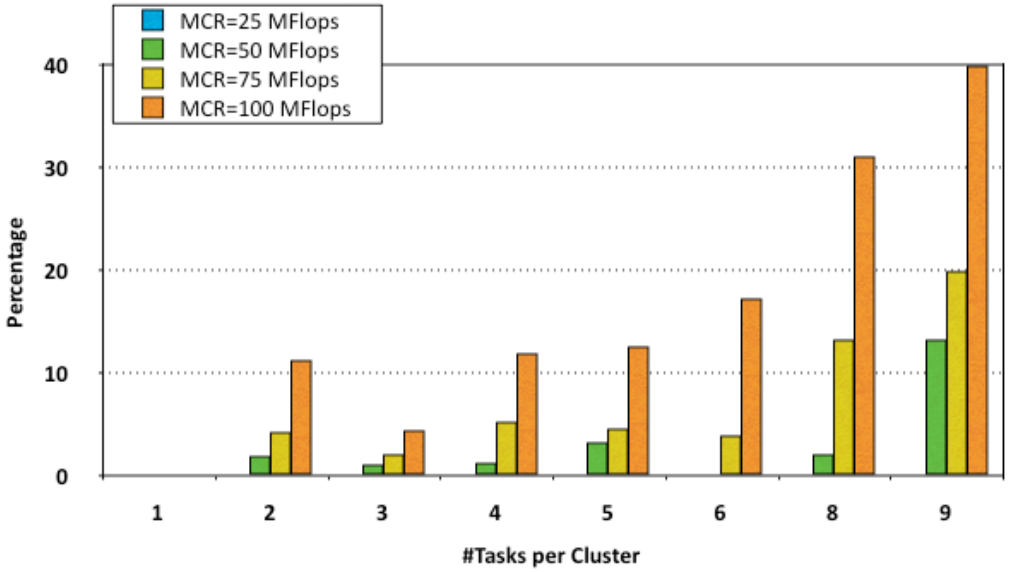

Fig. 5.4. Percentage of scheduling failures (2)

In the following evaluations, the average values are computed discarding the simulations resulting in scheduling failures.

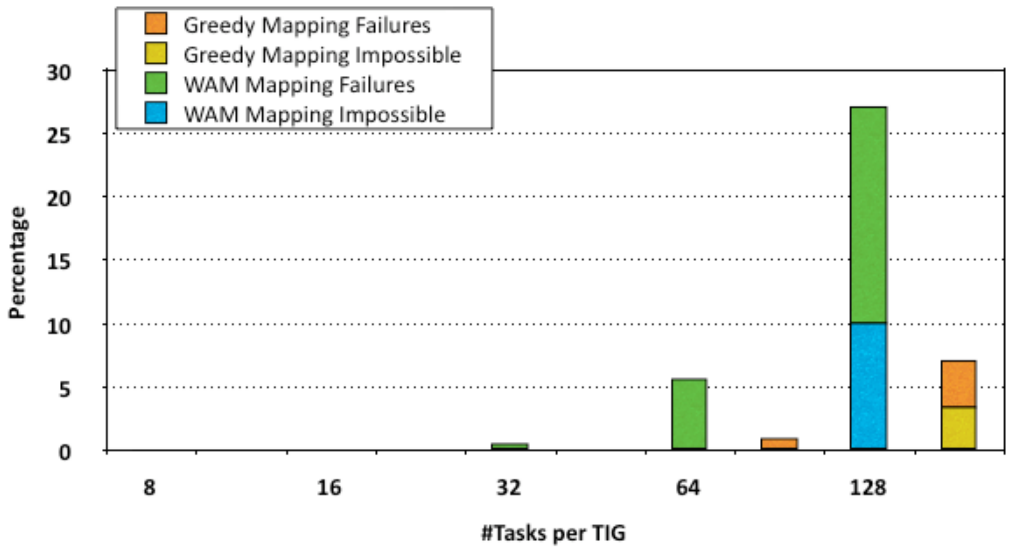

Fig. 5.5. Comparison of percentage of scheduling failures

In Figure 5.6 the average scheduling times of the proposed algorithm that was needed for conducting the simulation on the aforementioned resource are shown. It grows exponentially with the number of components per TIG. In Figure 5.7 these times are compared with those obtained by a greedy algorithm on the same simulations.

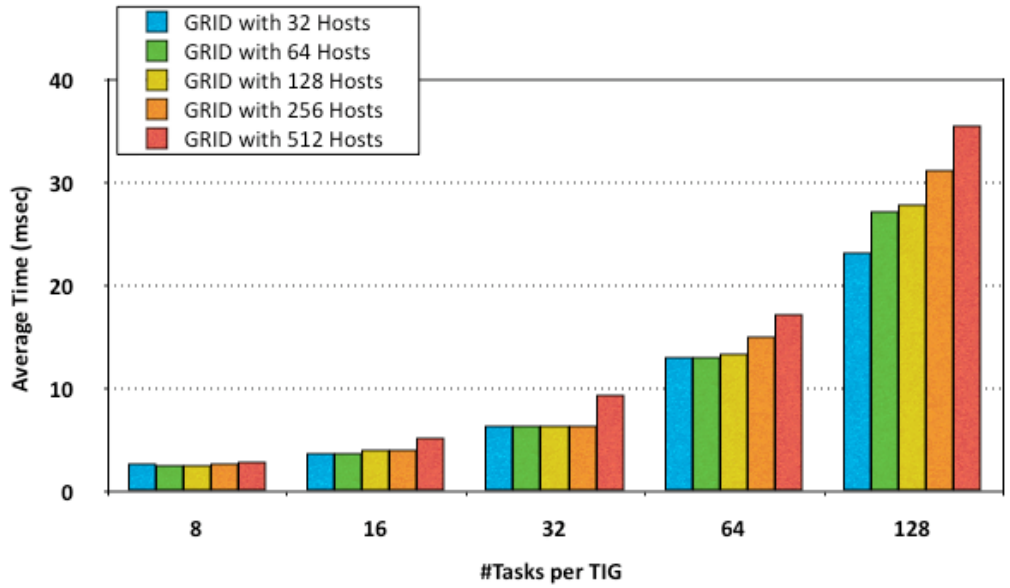

Fig. 5.6. Average scheduling time

It can be seen that the proposed algorithm presents better values than the greedy heuristics. It is mainly due to the fact that the greedy one must find the "best fit" on the

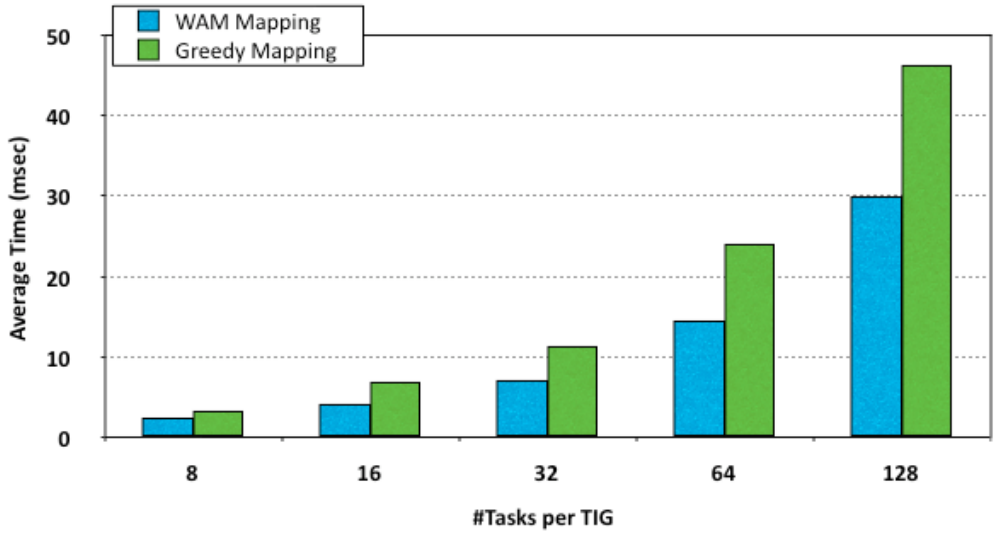

Fig. 5.7. Comparison of average scheduling times

queue of the resources for each components, while the other just tries to exploit user LAN resources. This is shown in Figure 5.8. For realistic Grid systems as the ones generated by Tiers, the proposed algorithm was always able to schedule tasks in the first suitable LAN or in LANs belonging to the same MAN, efficiently exploiting the Grid communication resources.

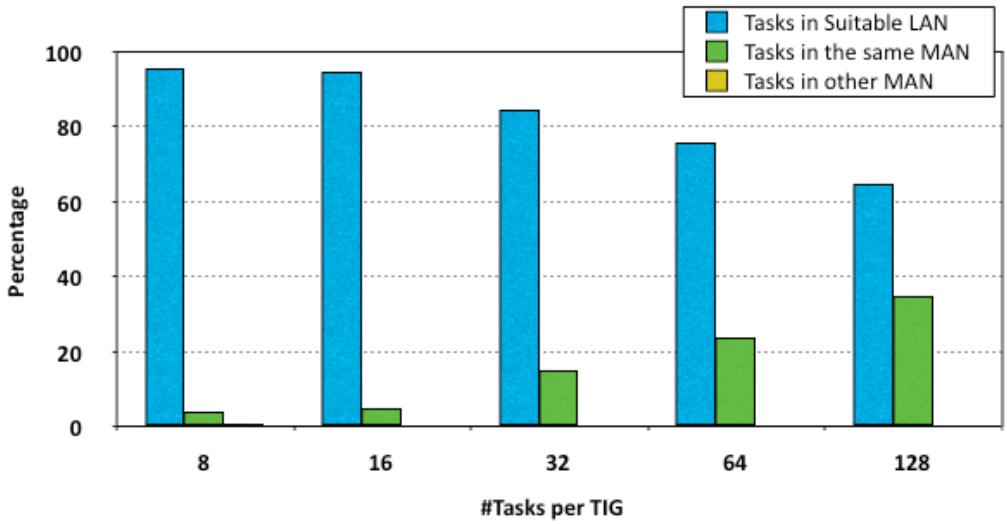

Fig. 5.8. Scheduling of TIG (4 hosts per LAN)

The LAN hit ratio is the percentage of the tasks of a cluster that are allocated into the suitable LAN. Since the allocation of the tasks in the same LAN allows to obtain a

better communication quality, higher values of the LAN hit ratio mean an improvement of the application throughput.

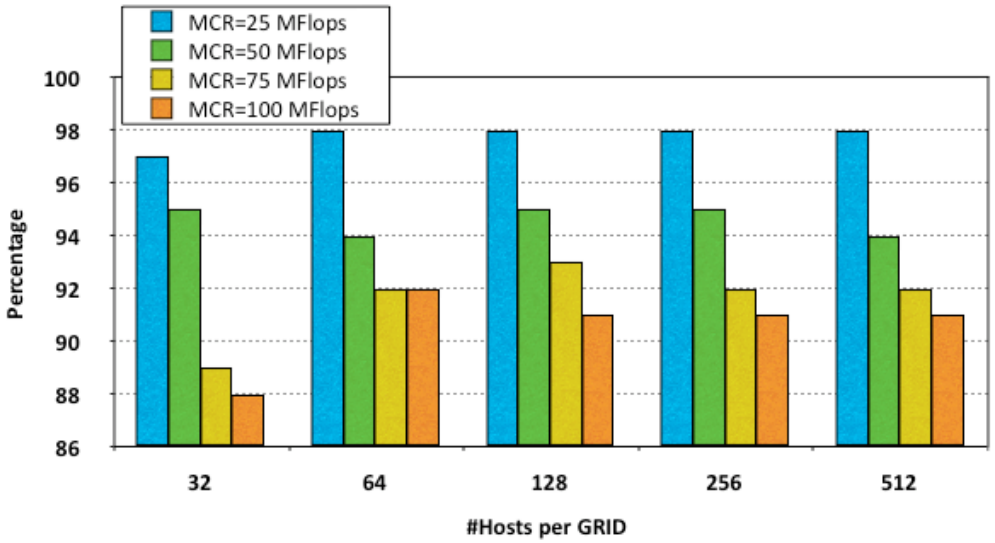

Fig. 5.9. Average LAN Hit Ratio (WASE)

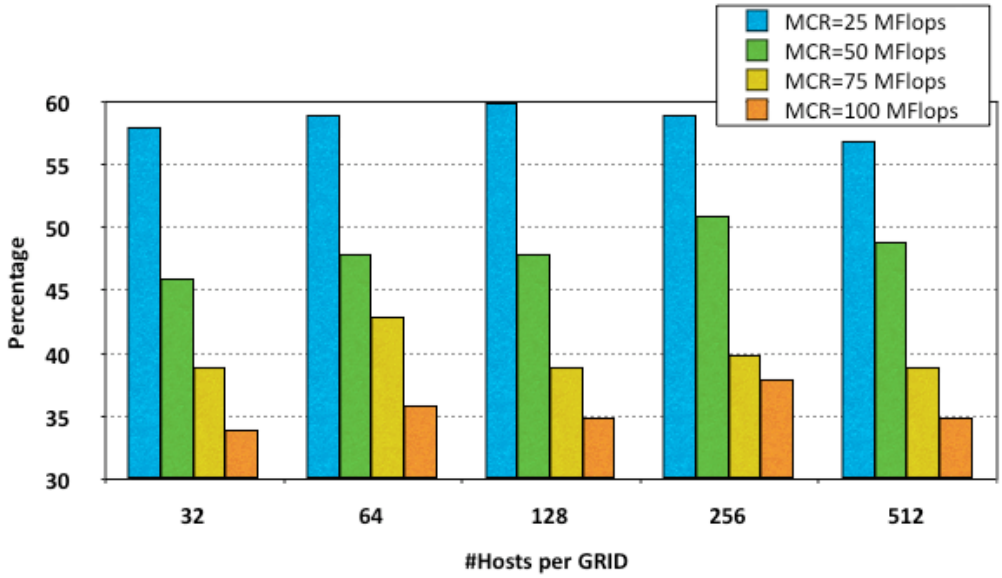

Fig. 5.10. Average LAN Hit Ratio (Greedy)

Figure 5.9 and Figure 5.10 show the results of the average LAN hit ratio using proposed and greedy algorithms, respectively. In such tests it can be observed that the LAN hit ratio of the proposed algorithm is around 90%, while the one of the greedy algorithm

spans from 20% to 60%. This result demonstrate that the proposed algorithm is able to allocate the tasks of a cluster in the same LAN and this allows to avoid a degradation of the communication quality.

Finally, Figure 5.11 shows a limit of the proposed solution: changing the size of the Grid, the minimum task-machine affinity does not change. This is due to the fact that the algorithm always search for good resources starting from the user LAN and moving in nearby LANs/MANs. This behavior prevents the algorithm to explore distant LANs, even if such LANs are able to host a whole cluster of tasks.

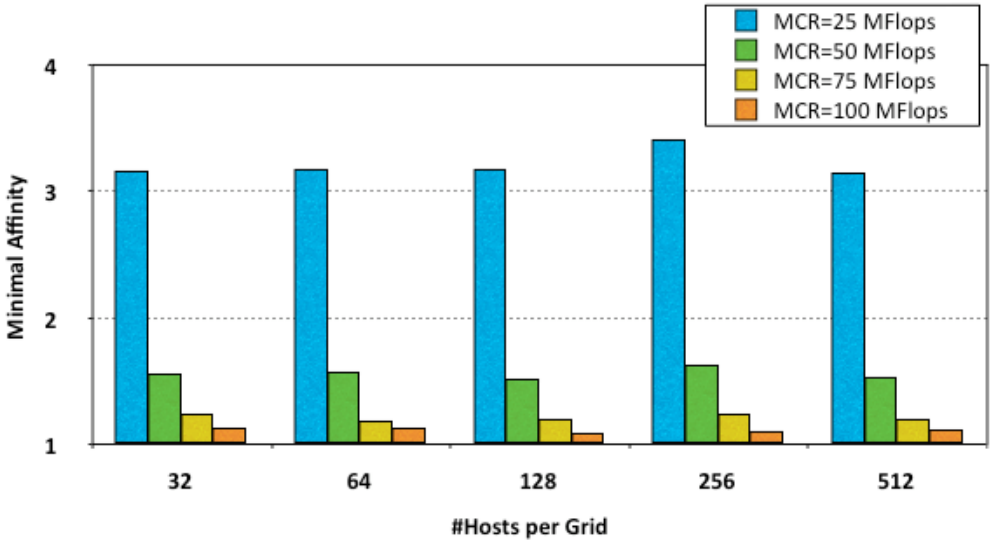

Fig. 5.11. Minimum Task Machine Affinity (TIGs with 8, 16, 32, 64 and 128 tasks)

5.5.4 Validation

Figure 5.12 shows the structure and the weights of the video rendering application used in this experiment. The weights have been collected through a short profiling execution of the application on a reference architecture.

In Figure 5.13 the testbed used to evaluate the algorithm is shown. It is a Grid composed by 23 heterogeneous computational resources (7 workstations and 2 clusters) distributed in three LANs connected through 1Gb/s optical links. The nominal computational power of each machines is listed in Table 5.1. The average initial load on each one has been set to 30%.

The clustering algorithm has been applied with three different similarity thresholds to obtain different clusters as shown in Figure 5.14. Then, the proposed algorithm has been applied with three different QoS levels, always considering ISTI as the user LAN. The allocation results, obtained as function of the number of clusters, are shown in Table 5.2.

It can be seen that increasing the QoS level causes a decrease in the average task-machine affinity. This is due to the increase of the computational power requested by the

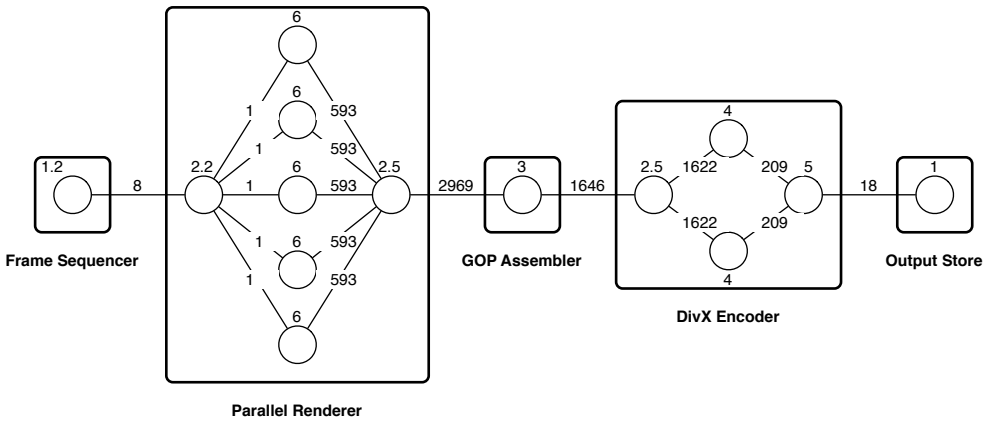

Fig. 5.12. Graph of the render-encode application

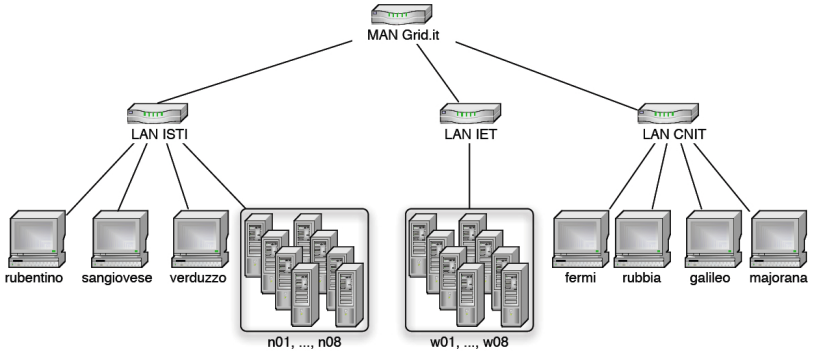

Fig. 5.13. Graph of the real Grid testbed

Resource	P_i (MFlop/s)	Resource	P_i (MFlop/s)
rubentino	1000	w01	650
sangiovese	950	w02	625
verduzzo	1000	w03	700
n01	1350	w04	650
n02	950	w05	1250
n03	950	w06	1300
n04	950	w07	1100
n05	950	w08	700
n06	950	fermi	1350
n07	950	galileo	1350
n08	950	rubbia	1300
		segre	1325

Table 5.1. Computational power of resources in the Grid testbed

tasks within a cluster. Moreover, the more the number of clusters increase, the more the inter-LAN communications increases (more clusters, more edges on the cuts). This is a kind of optimality parameter for the results of the allocation phase. In fact, consider the clustering result as the best one for the application. If the scheduling phase is able to

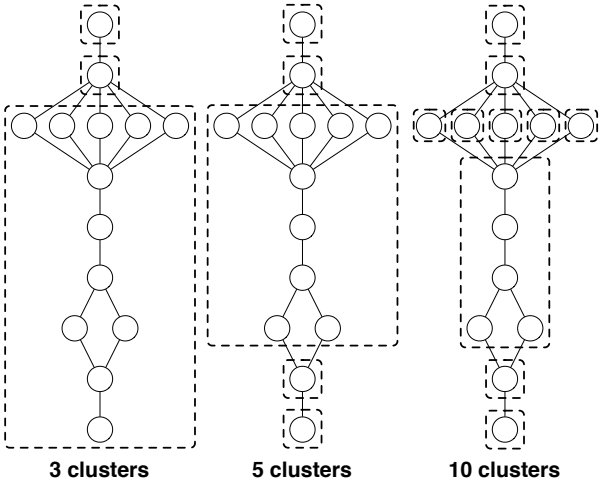

Fig. 5.14. Different clustering results of the test application

	MCR	LAN ISTI	LAN CNIT	LAN IET	Average Affinity
3 clusters	50	14	0	0	13.35
	100	14	0	0	5.63
	150	8	1	5	4.32
5 clusters	50	14	0	0	13.93
	100	14	0	0	6.72
	150	6	1	7	4.48
10 clusters	50	14	0	0	14.75
	100	14	0	0	7.00
	150	9	1	4	4.82

	Inter LAN Communication (clustering)	Inter LAN Communication (scheduling)
3 clusters	0.12%	0.00%
		0.00%
		20.81%
5 clusters	3.98%	0.00%
		0.00%
		24.48%
10 clusters	30.27%	0.00%
		0.00%
		26.04%

Table 5.2. Allocation results for the use case application

schedule every single cluster completely onto a single LAN, eventually we cannot obtain an inter-LAN communication value worse than the previous one. During the scheduling phases, the inter-LAN communication values are influenced by two effects:

1. a single cluster may be too "large" to be scheduled on a single LAN, and it must be split across several LANs. It happens with few and big clusters, and this effect worsen the final inter-LAN communication value;
2. several clusters may be scheduled on the same LAN, if there are available resources, improving the final inter-LAN communication value.

Both effects are present in the test results. In the case of three clusters, the scheduling algorithm is forced to split the big cluster in several LANs, while in the case of ten clusters, the heuristics is able to completely allocate the largest cluster on a LAN and to allocate several small clusters on the same LANs.

6

Component Applications Scheduling on Global Grids

In this chapter a launch-time scheduling heuristics is proposed and discussed. It aims to schedule component-based parallel applications on *global Grids*. The performance model developed in Chapters 3 and 4 is exploited to exploit a modified list heuristics (e.g max-max) including information on network bandwidths. In the case of global grids, although, the lack of structured information on the topology of the Grid and the high variability of the wide area networks characteristics make the qualitative approach developed in Chapter 5 unfeasible. To take care of this problem a fast statistical analysis of the network characteristics is employed, in order to identify clusters of resources characterized by similar, high-performance network bandwidths.

The content of this chapter has been presented in [134, 135].

6.1 Introduction

To fully exploit the network bandwidth increase which characterizes current computational Grids, several applications like parallel communication intensive and multimedia applications require the definition of alternative models and scheduling heuristics. The goal of these heuristics is to map a set of highly interacting tasks to a set of Grid resources connected by an acceptable bandwidth communication. For this reason, a suitable model of the Grid is needed as well. A Grid interconnection structure may include links with heterogeneous performances, hence no constraint on the network topology can be assumed. A Grid interconnection structure may include links with alternative performances, ranging from high speed optical ones connecting MANs to relatively slow links connecting different LANs within a MAN. Hence, no constraint on the network topology can be assumed and a hierarchical model is not appropriate.

The scheduling algorithm proposed in this chapter (QLSE) considers applications described by a TIG whose nodes and edges are labeled according to the Quality of Service requirements of the application. Node and edge labels describe, respectively, the Minimal Computational Requirement of the corresponding task and the Minimal Bandwidth Requirement of the corresponding communication. The Grid is modeled as a graph whose nodes and labels correspond to, respectively, LANs and connections among LANs. To avoid the resource aging phenomena, QLSE maps at launch time an application by applying a modified list heuristics exploiting dynamic information about the Grid status

returned by typical Grid monitoring tools. QLSE detects subsets of LANs connected by threshold values of communication bandwidth by clustering the Grid graph. These thresholds are defined by a statistical analysis of the QoS of the communications required by the user. QLSE orders the tasks of the applications according to a priority which takes into account both their computational requirement and their bandwidth requirements. The clusters and the LANs within each cluster are ordered on the basis of a combination of their computational power and communication bandwidth. Then, a max-max scheduling algorithm is applied to the resulting lists.

6.2 Application Model

The application model presented in Chapter 5 is exploited. The application graph $G_{App} = (N, V)$ is a weighted Task Interaction Graph (TIG) (see Fig. 6.1), where a vertex $n_i \in N$ models a computation, and a undirected edge $a_{ij} \in V$, with $i \neq j$, models a communication occurring between vertex n_i and vertex n_j. Vertices have an associated weight w_i representing the minimum computational bandwidth to execute the computation respecting the associated performance requirement, and edges have an associated weight w_{ij} representing the amount of data exchanged between two vertices in the time unit.

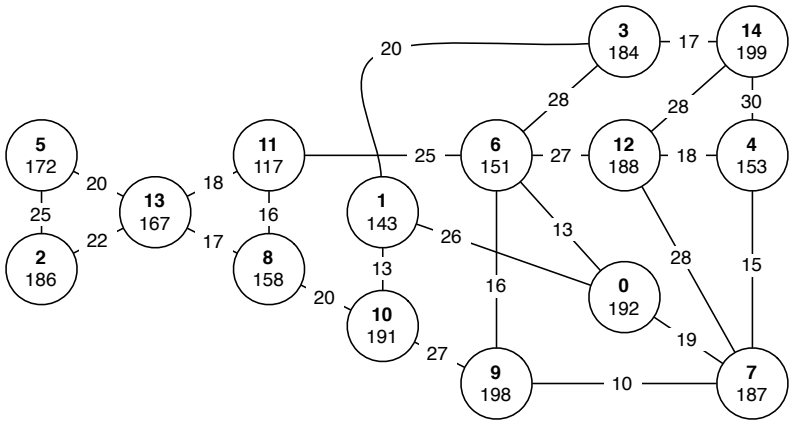

Fig. 6.1. Example of a weighted Task Interaction Graph with 14 nodes

6.3 Platform Model

A Grid is modeled as a set of computational resources geographically dispersed and interconnected through an interconnection network, e.g. Internet (a *global Grid*). To model this network, each resource is connected to a single, well known Local Area Network (LAN). A LAN is shared between the interconnected resources (i.e. shared bus), and it is characterized through its nominal communication bandwidth, expressed in Mb/s. Several

LANs are interconnected through an unreliable network whose topology is unknown. Exploiting network monitoring tools it is possible to estimate the bandwidth between two separate LANs. However, it is not viable to monitor every single communication path, and some paths may not exist or have a very limited bandwidth (e.g. smaller than 1 Mb/s). Nevertheless, without loss of generality, a connected topology is considered.

For each LAN, it is we assumed to know the following information: the number of hosts connected to the LAN and their instantaneous computational bandwidth[1] and the inner communication bandwidth of the LAN. An example of a Grid model is shown in Fig. 6.2. Note that if there is not a direct connection between two LANs, they can not communicate, even if there is a path between them composed by more than one edge. This assumption has a strong impact on the scheduling of two communicating components. In fact, they are forced to be mapped on the same LAN or onto two directly connected LANs.

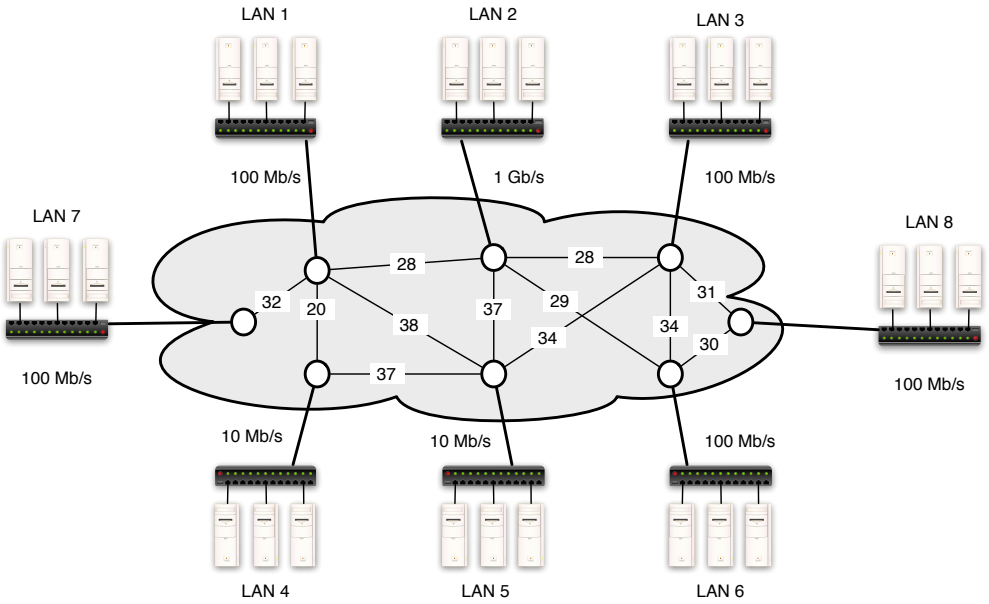

Fig. 6.2. Example of a modeled Grid with bandwidth information

6.4 Algorithm Architecture

The proposed algorithm takes as input the weighted graph of an application G_{App} and the weighted graph of the target Grid G_{Grid}. The algorithm's goal is to find a schedule of the application tasks to the Grid hosts in a way to fulfill simultaneously the minimum computational requirements of tasks and the minimum bandwidth requirements of communications.

[1] Each computational resource supports a multiprogramming environment.

The algorithm exploits a list scheduling heuristics which, by "appropriately" ordering the tasks, LANs and hosts, tries to meet the application QoS requirements by scheduling communicating tasks on the same LAN or on directly connected LANs.

6.4.1 Tasks Ordering

Task are ordered according to a *score* assigned to each of them. This score takes into account the weight of each node and it incorporates information on the topology of the application in order to exploit the high communication bandwidth of a LAN:

$$\forall n_i \in N, \quad score_i = w_i + \sum_{a_{ij} \in E} \left(\alpha_T w_{ij} + \beta_T w_j \right) \tag{6.1}$$

where α_T is a conversion factor and $\beta_T \in [0, 1]$ is a relative weight of the communicating tasks' computational requirements. This means that the score of a node is the sum of its weight, its complete communication requirement (the sum of edge weights) and a percentage of the weights of the nodes which it is communicating with. After several experiments to tune such parameters, the best results have been obtained with $\alpha_T = 1$ and $\beta_T = 0.25$.

The tasks are then structured in layers. The highest score task is placed in the first layer. The tasks directly connected to tasks in the previous layer are in the subsequent layer. The tasks in each layer are then grouped in subsets of communicating tasks; i.e. tasks directly connected in the TIG are put in the same subset. Then the subsets are ordered in decreasing order of number of tasks, and, inside a subset, tasks are ordered in descending order of score. The final task ordering is built considering the task in the first position, followed by the ordered tasks of the second layer and so on. In Fig. 6.3 an example in which the TIG of Fig. 6.1 structured in layers is shown, where the dashed lines identify the node subsets.

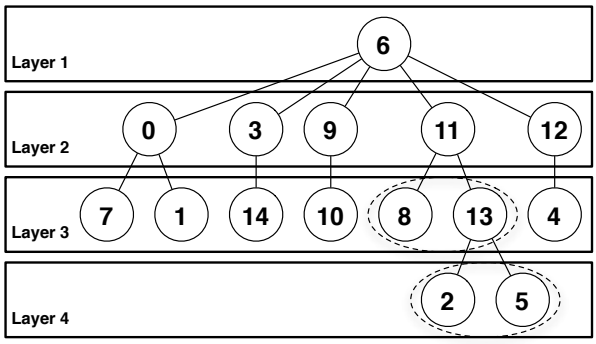

Fig. 6.3. Layered ordering of tasks of the TIG in Fig. 6.1

According to this ordering, we will try to allocate communicating tasks in the same LAN or in directly connected LANs. A problem may arise when dealing with cycles in the application graph. With the proposed ordering, we have no guarantee that the first and the last allocated tasks of a cycle will be allocated in the same LAN or in

directly connected LANs, thus generating a schedule that could be unable to fulfill the communication requirements. To solve this issue, we force all the cycle's tasks but the first to be considered part of the same subset of tasks.

6.4.2 Hosts Ordering

Each host in a LAN shares the same communication medium, so in a LAN the hosts can be ordered in decreasing order of computational power. The priority of a LAN is calculated similarly to (6.1):

$$\forall L_i \in N_{LAN}, \quad prio_i = \sum_{h_{ij} \in R_i} r_{ij} + \sum_{(ij) \in E_{LAN}} \left(\alpha_L bw_{ij} + \beta_L \sum_{h_{jk} \in R_j} r_{jk} \right) \tag{6.2}$$

where N_{LAN} is the set of the Grid LANs, R_i is the set of hosts connected to the LAN L_i, h_{ij} is the j^{th} host of the LAN L_i and r_{ij} is its computational power and bw_{ij} is the average bandwidth of the link between L_i and L_j. E_{LAN} represents the set containing the edges of the graph modeling the connections between the LANs (see Fig. 6.2):

Also in this case, α_L is a conversion factor and $\beta_L \in [0, 1]$ is a relative weight of the total power of the directly connected LANs. Again, the best results are obtained choosing $\alpha_L = 1$ and $\beta_L = 0.25$.

6.4.3 LAN Clustering

LAN to LAN interconnections are unreliable. To try to guarantee the required bandwidth performance during the execution of the application, we would like to exploit LAN to LAN interconnections with the highest communication bandwidths. To do so, we can, in first instance, ignore the low bandwidth links of the Grid. However we have no guarantees that this link removal will allow us to find feasible solutions. Anyway, we can proceed with some "tentative mappings", starting from a Grid graph with only the highest communicating edges. Then, if no solution is admissible, we add some other edges, and we continue until, at the end, we consider the original Grid graph.

After several tries, we have decided to use the *quartiles* of the links bandwidth distribution to set the values of the edges to be removed in each step. In Fig. 6.4 an example of bandwidth distribution and its quartiles is shown.

In the first tentative mapping, we consider the $x_{1.00}$ quartile, hence removing every LAN to LAN interconnection from the Grid graph. In doing so, we are looking for a single LAN able to hosts the whole application. In the second tentative mapping, we remove from the Grid graph the edges with bandwidth values smaller than the upper quartile $x_{.75}$. In doing so, we may obtain several disconnected LAN clusters. In such case, we order the clusters and we try to find a solution on a single cluster. If no cluster allowing an admissible value is found, we repeat the procedure with the middle quartile $x_{.50}$, then with the lower quartile $x_{.25}$ and eventually with the original Grid graph (in general we can assume $x_{.00} = 0$).

The clusters are ordered in decreasing order of their priority. The priority of a cluster is computed as the sum of the computational and communication bandwidths of the cluster LANs and the communication bandwidths between the LANs of the cluster.

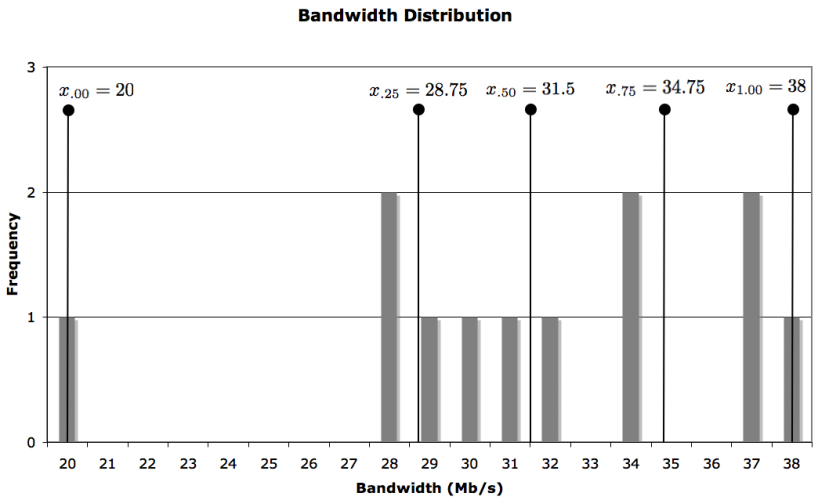

Fig. 6.4. Bandwidth distribution of Grid in Fig. 6.2 and its quartiles

6.4.4 Component Allocation

When a component and a LAN are selected, the allocation of the selected component on the selected LAN can be performed if the following *allocation conditions* are met:

- The LAN contains at least one host with computational power greater than or equal to the component weight.
- The sum of the weights of the edges between the selected component and the communicating ones already allocated on the selected LAN must be smaller than or equal to the LAN bandwidth.
- The sum of the weights of the edges between the selected component and the communicating ones already allocated on another LAN must be smaller than or equal to the bandwidth between the two LANs.

If such conditions are met, the component is allocated to the LAN and the computational power of the selected resource is decreased by the task's weight, the LAN bandwidth is decreased by the sum of the weight of the edges between the component and the already allocated ones and the LAN-LAN bandwidths are decreased similarly.

6.4.5 Scheduling Heuristics

According to the previous considerations, the proposed scheduling heuristics performs the following steps:

6.5 Performance Evaluation

The evaluation of the scheduling solutions carried out by QLSE was conducted by simulations applying the algorithm to a suite of task graphs and Grid platforms randomly generated.

Input: Application Graph G_{App}, Grid graph G_{Grid}
Output: A feasible allocation of components to resources or an error message

```
1  Compute the score of each node of G_App according to (6.1);
2  Build the hierarchical structure of G_App eliminating cyclic paths;
3  Build the ordered array A_1 of components;
4  Compute the priority of each LAN according to (6.2);
5  Compute the quartiles x_1.00, ..., x_.00 of G_grid;
6  foreach (quartile (from x_1.00 to x_.00)) do
7      The Grid graph is clustered;
8      The ordered array A_2 of clusters is built;
9      foreach (cluster in A_2) do
10         The ordered array A_3 of LANs is built according to their
           priorities;
11         Copy locally A_1 to A_1^tmp;
12         foreach (component in A_1^tmp) do
13             c = A_1^tmp.remove();
14             Search in A_3 the first LAN l meeting the allocation
               conditions in 6.4.4 for component c;
15             if (l ≠ ∅) then
16                 Allocate c on l decreasing computational power and
                   bandwidths;
17             else
18                 Restart with the next cluster;
19             if (A_1^tmp.empty()) then
20                 return allocation;
21         if (an allocation has been found) then
22             return allocation;
23         else
24             Restart with the next quartile;
25 return error("Unable to find a feasible scheduling");
```

Algorithm 5: The QoS List Heuristics

Tables 6.1 and 6.2 show the bounds values defined for each parameter used by a uniform distribution to generate TIGs and Grids, respectively. The use of this distribution is due to the lack of experimental data sets on TIGs and Grids. It is assumed that the use of such distribution will provide a more uniform coverage of cases. TIGs and Grids with three different sizes were used in each test. To obtain valid statistical values, for all the 2187 combinations obtained from the TIGs and Grids parameters, 10 TIGs and 10 Grids were randomly generated to run a test. In total, 21870 test cases were generated.

To validate and evaluate QLSE we implemented a greedy scheduling algorithm, and we applied it to every test case. This algorithm simply orders the TIG's tasks queue in decreasing order of MCR and the LANs queue in decreasing order of aggregate computational power. The machines belonging to a LAN are ordered in decreasing order of

Size	# Components	Node weights (MFlop/s)	Edge weights (Mb/s)
Small	8 – 32	100 – 200	10 – 30
Medium	32 – 48	200 – 400	20 – 40
Large	48 – 64	400 – 600	30 – 50

Table 6.1. Parameters used to generate TIGs

Size	# LAN	# Resources per LAN	Resource Power (MFlop/s)	Inter-LAN bandwidth (Mb/s)
Small	8 – 16	4 – 12	600 – 800	20 – 40
Medium	12 – 24	8 – 24	800 – 1000	30 – 50
Large	16 – 32	16 – 48	1000 – 1300	40 – 60

Table 6.2. Parameters used to generate Grids

computational power. The algorithm performs a *best fit* heuristics to map the queued tasks on the LANs' hosts.

The validation of QLSE was based on the criterion that it would be able to carry out at least a feasible solution, i.e. a solution that satisfies the MCRs and MBRs required by an application, when at least a valid solution exists for a TIG schedule. This criterion can be deterministically evaluated generating TIGs and Grids in such a way that a valid solution exists. Figure 6.5 shows the average percentage of scheduling failures obtained by both algorithms w.r.t. the number of tasks per TIGs. A scheduling failure occurs when the algorithm is not able to map a TIG satisfying both MCRs and MBRs requirements. An increment of the number of tasks causes an almost linear increment of the scheduling failures obtained by the Greedy algorithm, while QLSE is able to carry out almost all the valid solutions.

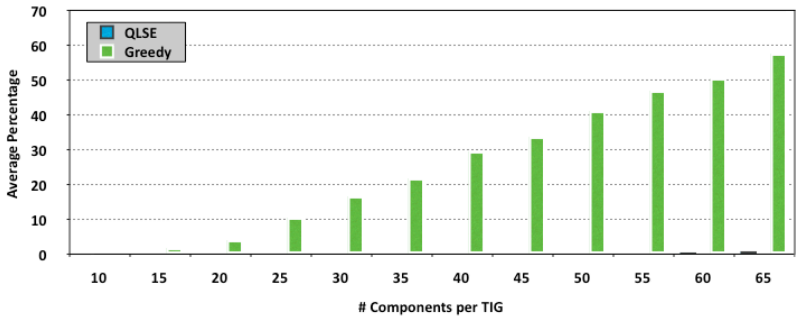

Fig. 6.5. Average percentage of scheduling failures

To evaluate the scheduling carried out by QLSE we exploited different criteria: the LAN hit ratio, the task-machine computational ratio, and the average scheduling time. The LAN hit ratio returns the percentage of communications allocated on the same LAN. It is computed as the ratio between the sum of TIG's MBRs associated to the communicating tasks mapped in the same LAN and the TIG's MBRs' sum. An higher LAN hit ratio corresponds to an higher throughput between communicating tasks increases.

The task-machine computational ratio measures how the more powerful machines are exploited to run TIG's tasks. An higher task-machine computational ratio corresponds to an higher application throughput. It is defined as the ratio between the computational power of the machine where a task is mapped and the task's MCR. The average scheduling time of QLSE is exploited to evaluate the effectiveness of the algorithm with respect to the aging effect of the resources. To compute these performance parameters values only the simulations that carried out a feasible solution by both algorithms were considered.

Figure 6.6 shows the results of the average LAN hit ratio, expressed in percentage, w.r.t. the number of tasks per TIG, using QLSE and the greedy algorithm.

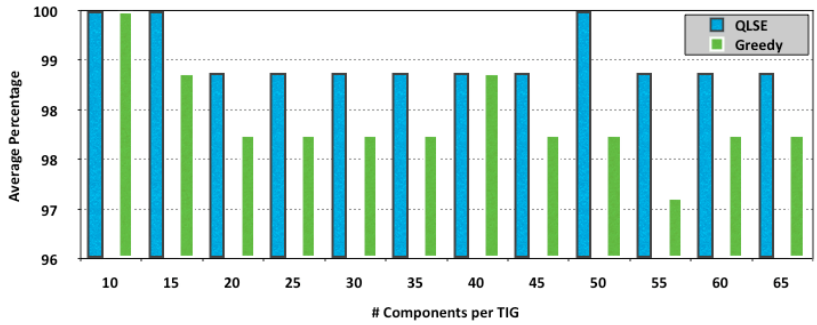

Fig. 6.6. Average LAN Hit Ratio

In these tests both algorithms obtain a LAN hit ratio very close to 100%. We can observe that QLSE is always able to obtain better values than the greedy algorithm, since it is able to allocate in the same LANs the tasks with an higher degree of communication.

Figure 6.7 shows the average component-resource computational ratio w.r.t. the number of tasks per TIG.

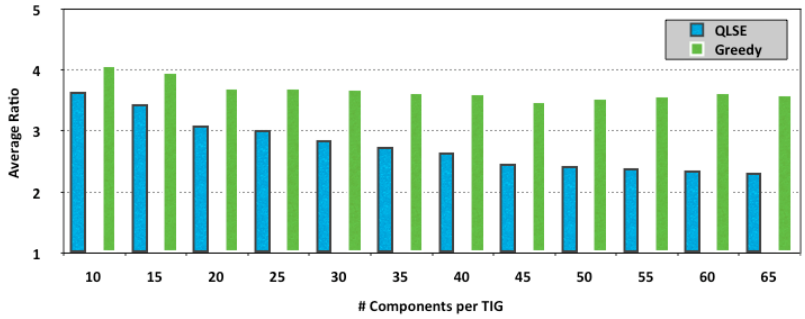

Fig. 6.7. Average component-resource computational ratio

It can be observed that QLSE is able to map the tasks on machines with a computational power 2 or 3 times greater than their MCR. The Greedy algorithm obtained a average task-machine computational ratio always greater than QLSE, because it selects first the LANs with a greater aggregate computational power, thereafter it exploits for

the valid solutions a smaller number of LANs (no more than 2 in our test cases) than QLSE to map a TIG. However, by exploiting appropriate inter-LANs links, QLSE is able to guarantee the communication QoS (i.e. the required TIG's MBRs) also when TIG's tasks are mapped on more LANs.

The scheduling time grows almost linearly with the number of tasks per TIG in both algorithms. Differences between the two algorithms are appreciable starting from about 50-nodes TIGs. Scheduling 64-nodes TIGs, QLSE requires an execution time10% greater than the one required by the greedy solution. This is obviously due to the higher complexity of QLSE, which, to be able to find a valid solution in almost all test cases, analyses the inter-LAN links capabilities more accurately than the greedy algorithm.

Figure 6.8 shows the results of the average scheduling time, expressed in msec. It can be seen that the scheduling time grows with the number of tasks per TIG. Differences between the two algorithms are appreciable starting from about 50-nodes TIGs.

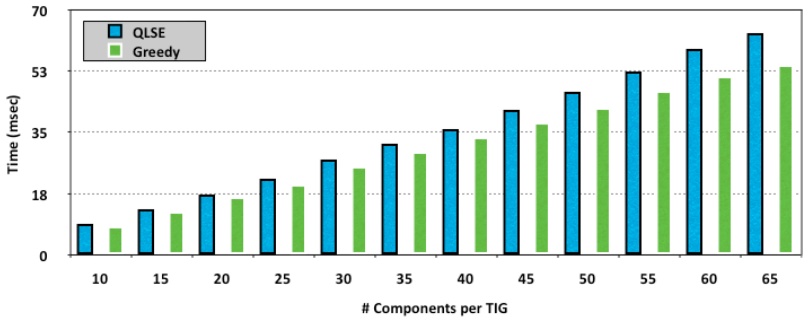

Fig. 6.8. Average scheduling times

These results have been obtained running the algorithms on a Pentium4 processor with a clock frequency of 3.0GHz and a memory of 1GB.

7

Conclusions

7.1 Summary

This thesis investigated three key requirements for the developments of Grid applications: the need for a performance model of such applications, the need for a formal specification of a performance contract and the need for scheduling strategies to map such applications on different Grids, exploiting the information provided by models and contracts.

More precisely, the decision to investigate component-based applications has been motivated by the current research trends in Grid programming model. A dynamic model for applications interacting through streams of data, processed by several autonomous software components is presented. This model has been exploited to describe the structural behavior of a large class of applications in a general way. This description is completely independent from the runtime target platform. Then a formal methodology has been introduced to specify performance models and contracts for hierarchical component-based applications. This thesis presented an algorithm to translate a user-specified performance contract into elementary resource requirements exploiting component annotations. The suitability of this approach to complete automatization has been discussed, and the evaluation of the whole methodology with a real application deployed on homogeneous and heterogeneous sets of resources has been evaluated. Exploiting the proposed performance model and contract such applications have been described with task interaction graphs, whose nodes and edges are labeled according to the QoS requirements derived through the performance contract resolution algorithm. Then two launch time scheduling heuristics have been proposed to map parallel component-based applications on specific Grid platforms. The effort needed devise an almost optimal initial scheduling of an application on a set of dynamic resource may not pay, mainly because the dynamicity can invalidate the (sub-)optimality of the proposed solution in short times. Then two scheduling heuristics have been designed, both being able to quickly identify feasible initial scheduling solutions dealing with computational and communication requirements. The first heuristics (Wide Area Scheduling hEuristics, WASE) targets hierarchically structured Grids exploiting qualitative information on the application structure, while the second heuristics, (QoS List Scheduling hEuristics, QLSE) targets unstructured global Grids trying to exploit quantitative information on the application structure. Both heuristics try to schedule a parallel application satisfying the minimal computational requirements of its tasks, and exploiting the intra-LAN communications in order to maximize the throughput between

communicating components. WASE and QLSE pre-process the application model and/or the Grid model to find clusters requiring/providing high communication characteristics. Then the allocation strategy is carried out by exploiting a priority-based technique to satisfy the task minimum computational requirement. The heuristics try to allocate tasks in resource clusters, in order to maximize the throughput between communicating tasks. In order to evaluate the quality of the scheduling algorithm, several tests were conducted by simulating the execution of a large number of parallel applications on a number of Grids both randomly generated. Moreover, such analysis compared the proposed algorithm with a greedy one, which exploits a best fit heuristics. This simple algorithm has been chosen mainly because it is the faster and simpler available algorithm compared to the proposed heuristics. The tests demonstrated that the presented algorithms were able to carry out a valid solution (i.e. satisfying the QoS) in a good percentage of the simulated test cases. Moreover, the high values obtained for both the tasks' communication and computation throughputs demonstrated the applicability of the approach. Eventually, the LAN hit-ration performance was introduced, to evaluate the objective to keep most communications in clusters of resources with high communication bandwidth available, and very good results have been obtained.

Nevertheless, the presented performance model is not general, for two reasons. First, communications with stream semantics were modeled, because a large number of Grid applications use these mechanisms for communications. However, several distributed applications exploit the Remote Procedure Call (RPC) semantics. These applications, and those exploiting mixed semantics, are not covered by the presented model.

Second, the ergodicity condition imposed on communications to derive the steady state model limits the number of stream-based applications it can be applied to. For example, several data-mining applications are composed by module whose execution times does not depend only on how many items they receive, but on their values as well.

Moreover, annotations still have to be inserted in component descriptions by their developers, along with their basic performance characteristics. Such measures can be difficult to obtain, and the whole methodology strongly depends on them.

Concerning the scheduling heuristics, the initial mapping should 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. The presented approach must then be coupled with rescheduling strategies at runtime to solve such problems. Nevertheless, the presented performance model/contract can be exploited at runtime 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.

7.2 Research Directions

The presented work has several interesting research directions as improvements or extensions. The proposed performance model is, to the best of the author's knowledge, the first attempt in giving an analytical performance characterization to component-based

distributed applications. Although some required constraints are difficult to remove (e.g. ergodicity), some choices can be relaxed. The next logical step is to remove the limit on the stream semantics. RPC-like communication semantics are exploited in several distributed programming frameworks, such as, for example, the asynchronous future calls semantics present in the ProActive framework and its implementation of the Fractal component model. In fact, component interfaces with this semantic returning void types can be seen as communications with a lazy stream semantics.

To fully understand the potentiality of the presented model, a sensitive analysis of its robustness should be performed. The applicability of the model relies on two basic assumptions: the correctness of the structural behavior specification and the reliability of the performance annotations. The effect of quantitative errors in such assumptions has not been evaluated, but it is mandatory in the next future to understand the impact of such errors on the runtime performance. The information provided by this analysis might give indications on improvements to the constraints resolution algorithm.

Furthermore, the component developer must still provide performance annotations. Besides architectural and measurement peculiarities, a general performance evaluation procedure for components has not been proposed. This procedure is, in general, difficult and complex, but components have a structure that could be exploited to make simplifying assumptions and build standard performance evaluation/monitoring mechanisms.

The adoption of the proposed performance model/contract should simplify some tasks that are common in high performance computing for Grids. As an example, this approach can be exploited to design and implement middleware services targeting: fault-tolerance (by means of data replication and/or checkpointing), runtime component instantiation and runtime reconfiguration.

Eventually, the presented heuristics can be further investigated in several directions. A first development should be to refine the suitability function to a better choice of the LAN for a group of communicating components. Such analysis might reduce the fragmentation effect of the tasks belonging to the same group mapped on different LANs. Another improvement should be the support of the dynamic spawning of new tasks of a running application. In fact, several parallel applications are able, at runtime, to carry out dynamic load-balancing by creating new components or by splitting a big component in several sub-components. Furthermore, during the evaluation phases some parameters have been chosen empirically. It is important to determine if such parameters have a general applicability or if they depend on the application/graph and how they could be determined in such case. With this analysis, the functionalities of the heuristics could be extended to new classes of parallel applications.

References

1. L. Smarr and C. E. Catlett. Metacomputing. *Communications of the ACM*, 35(6):44–52, 1992.
2. I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, 15(3):200–222, 2001.
3. M. I. Cole. *Algorithmic Skeletons : Structural Management of Parallel Computation*. PhD thesis, MIT, MA, October 1989.
4. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley/ACM Press, 2^{nd} edition, 2002.
5. M. Chetty and R. Buyya. Weaving Computational Grids: How Analogous Are They with Electrical Grids? *Computing in Science and Engineering*, 4(4):61–71, 2002.
6. I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1996.
7. D. D. Roure, M. A. Baker, N. R. Jennings, and N. R. Shadbolt. The Evolution of the Grid. In F. Berman, G. Fox, and T. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley and Sons Ltd, 2003.
8. J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003.
9. D. McIllroy. Mass Produced Software Components. In P. Naur and B. Randall, editors, *Software Engineering: Report on a Conference by the NATO Science Committee*, pages 138–155. NATO Scientific Affairs Division, Brussels, 1968.
10. Expert Group Report. Next Generation Grids 2: Requirements and Options for European Grids Research 2005-2010 and Beyond. `ftp://ftp.cordis.europa.eu/pub/ist/docs/ngg2_eg_final.pdf`, July 2004.
11. M. Aldinucci, M. Coppola, S. Campa, M. Danelutto, M. Vanneschi, and C. Zoccolo. Structured Implementation of Component-based Grid Programming Environments. In V. Getov, D. Laforenza, and A. Reinefeld, editors, *Future Generation Grids*, CoreGRID series, pages 217–239. Springer Verlag, November 2005.
12. E. Bruneton, T. Coupaye, M. Leclercq, V. Quèma, and J.-B. Stefani. The Fractal component model and its support in Java: Experiences with Auto-adaptive and Reconfigurable Systems. *Software Practice & Experience*, 36(11/12):1257–1284, 2006.
13. M. Aldinucci, A. Petrocelli, E. Pistoletti, M. Torquati, M. Vanneschi, L. Veraldi, and C. Zoccolo. Dynamic reconfiguration of grid-aware applications in ASSIST. In J. C. Cunha and P. D. Medeiros, editors, *Proc. of* 11^{th} *International Euro-Par 2005: Parallel and Distributed Computing*, volume 3648 of *Lecture Notes in Computer Science*, pages 771–781, Lisboa, Portugal, September 2005. Springer Verlag.
14. M. I. Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, March 2004.
15. J. W. Hooper and R. O. Chester. *Software Reuse: Guidelines and Methods*. Perseus Publishing, NY, 1991.

16. K. Crenecky and U. Eisenecker. *Generative Programming.* Addison-Wesley, MA, 2000.
17. G. Booch. *Object-oriented analysis and design with applications.* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 2^{nd} edition, 1994.
18. A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, Hewlett-Packard, 1994.
19. D. I. Joshi and P. A. Vorobiev. *JFC: Java Foundation Classes.* John Wiley & Sons, Inc., 1998.
20. Object Management Group. Common Object Request Broker Archictecture. `http://www.omg.org/docs/formal/04-03-12.pdf`, July 1995.
21. W. Grosso. *Java RMI.* O'Reilly Media, October 2001.
22. Object Management Group. The CORBA Component Model. `http://www.omg.org/docs/formal/06-04-01.pdf`, April 2006.
23. A. J. van der Steen. Issues in computational frameworks. *Concurrency and Computation: Practice and Experience*, 18(2):141–150, 2006.
24. R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a Common Component Architecture for High-Performance Scientific Computing. In *Proc. of* 8^{th} *IEEE International Symposium on High Performance Distributed Computing (HPDC99)*, pages 115–124, Washington, DC, USA, August 1999. IEEE Computer Society.
25. T. Dahlgren, T. Epperly, G. Kumfert, and J. Leek. Babel User's Guide. CASC, Lawrence Livermore National Laboratory, Livermore, CA, 2006.
26. D. Gannon, S. Krishnan, L. Fang, G. Kandaswamy, Y. Simmhan, and A. Slominski. On Building Parallel & Grid Applications: Component Technology and Distributed Services. *Cluster Computing*, 8(4):271–277, 2005.
27. A. P. W. Benjamin A. Allan, Robert C. Armstrong, J. Ray, D. E. Bernholdt, and J. A. Kohl. The CCA core specification in a distributed memory SPMD framework. *Concurrency and Computation: Practice and Experience*, 14(5):323–345, 2002.
28. F. Bertrand and R. Bramley. DCA: A Distributed CCA Framework Based on MPI. In *Proc. of* 9^{th} *International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 04)*, pages 80–89, Los Alamitos, CA, USA, April 2004. IEEE Computer Society.
29. M. Aldinucci, M. Coppola, M. Danelutto, N. Tonellotto, M. Vanneschi, and C. Zoccolo. High Level Grid Programming with ASSIST. *Computational Methods in Science and Technology*, 12(1):21–32, 2006.
30. M. Coppola, L. Presti, M. Pasquali, and M. Vanneschi. An Experiment with High Performance Components for Grid Applications. In T. Kielmann, E. Aubanel, V. C. Bhavsar, M. Frumkin, and R. F. van der Wijngaart, editors, *Proc. of* 10^{th} *International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 05)*, Washington, DC, USA, April 2005. IEEE Computer Society.
31. M. Aldinucci, M. Danelutto, A. Paternesi, R. Ravazzolo, and M. Vanneschi. Building interoperable grid-aware ASSIST applications via WebServices. In G. R. Joubert, W. E. Nagel, F. J. Peters, O. Plata, P. Tirado, and E. Zapata, editors, *Parallel Computing: Current & Future Issues of High-End Computing, PARCO 2005*, volume 33 of *NIC*, pages 145–152. Research Centre Jülich, December 2005.
32. K. Keahey and D. Gannon. PARDIS: CORBA-based Architecture for Application-level Parallel Distributed Computation. In *Proc. of the 1997 ACM/IEEE Conference on Supercomputing (Supercomputing 97)*, pages 1–14, New York, NY, USA, 1997. ACM Press.
33. A. Denis, C. Pérez, and T. Priol. Achieving portable and efficient parallel CORBA objects. *Concurrency and Computation: Practice and Experience*, 15(10):891–909, 2003.
34. A. Denis, C. Pérez, T. Priol, and A. Ribes. PADICO: A Component-Based Software Infrastructure for Grid Computing. In *Proc. of* 17^{th} *International Symposium on Parallel and Distributed Processing (IPDPS 03)*, Washington, DC, USA, 2003. IEEE Computer Society.
35. Basic Features of the Grid Component Model. CoreGRID's Programming Model Virtual Institute, `http://www.coregrid.net/mambo/content/blogcategory/13/292/`, September 2006.

36. M. Aldinucci, S. Campa, M. Coppola, M. Danelutto, D. Laforenza, D. Puppin, L. Scarponi, M. Vanneschi, and C. Zoccolo. Components for high performance Grid programming in Grid.it. In V. Getov and T. Kielmann, editors, *Proc. of the Workshop on Component Models and Systems for Grid Applications*, CoreGRID series, Saint Malo, France, January 2005. Springer Verlag.
37. R. Baraglia, M. Danelutto, T. Fagni, D. Laforenza, S. Orlando, A. Paccosi, N. Tonellotto, M. Vanneschi, and C. Zoccolo. HPC Application Execution on Grids. In V. Getov, D. Laforenza, and A. Reinefeld, editors, *Future Generation Grids*, CoreGRID series. Springer Verlag, November 2005.
38. M. Aldinucci, M. Danelutto, and M. Vanneschi. Autonomic QoS in ASSIST grid-aware components. In B. D. Martino and S. Venticinque, editors, *Proceedings of the* 14^{th} *Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP2006)*, pages 221–230, Montbéliard, France, February 2006. IEEE Computer Society Press.
39. S. Gorlatch and J. Dünnweber. From Grid Middleware to Grid Applications: Bridging the Gap with HOCs. In V. Getov, D. Laforenza, and A. Reinefeld, editors, *Future Generation Grids*, CoreGRID series. Springer Verlag, November 2005.
40. S. Gorlatch and J. Dünnweber. HOC-SA: A Grid Service Architecture for Higher-Order Components. In *Proc. of the 2004 IEEE International Conference on Services Computing (SCC 04)*, Sahngai, Cina, September 2004. IEEE Computer Society.
41. OASIS. Web Service Resource Framework Specification (WSRF). `http://www.oasis-open.org/committees/documents.php`, March 2006.
42. F. Baude, D. Caromel, and M. Morel. From Distributed Objects to Hierarchical Grid Components. In *Proc. of the International Symposium on Distributed Objects and Applications (DOA)*, volume 2888 of *Lecture Notes in Computer Science*, pages 1226–1242, Dresden, Germany, November 2003. Springer Verlag.
43. 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)*, pages 3–10, Toronto, Ontario, Canada, 2001. ACM Press.
44. L. G. Williams and C. U. Smith. PASA(SM): An Architectural Approach to Fixing Software Performance Problems. In *Proc. of* 28^{th} *International Computer Measurement Group Conference*, pages 307–320, Reno, Nevada, USA, 2002.
45. V. Cortellessa and R. Mirandola. PRIMA-UML: a performance validation incremental methodology on early UML diagrams. *Science of Computer Programming*, 44(1):101–129, 2002.
46. K. Kant. *Introduction to Computer System Performance Evaluation.* McGraw-Hill, 1992.
47. 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(1):20–34, 1995.
48. S. Bernardi, S. Donatelli, and J. Merseguer. From UML sequence diagrams and statecharts to analysable petrinet models. In *Workshop on Software and Performance*, pages 35–45, 2002.
49. P. J. B. King and R. Pooley. Derivation of Petri Net Performance Models from UML Specifications of Communications Software. In *Proc. of* 11^{th} *International Conference on Computer Performance Evaluation: Modelling Techniques and Tools (TOOLS 00)*, pages 262–276, London, UK, 2000. Springer-Verlag.
50. S. Gilmore and J. Hillston. The PEPA workbench: a tool to support a process algebra-based approach to performance modelling. In *Proc. of* 7^{th} *International Conference on Computer Performance Evaluation: modelling techniques and tools*, pages 353–368, Secaucus, NJ, USA, 1994. Springer-Verlag New York, Inc.
51. S. Gilmore, J. Hillston, L. Kloul, and M. Ribaudo. Software performance modelling using PEPA nets. In *Proc. of* 4^{th} *International Workshop on Software and Performance (WOSP 04)*, pages 13–23, New York, NY, USA, 2004. ACM Press.
52. C. U. Smith. *Performance Engineering of Software Systems.* Addison-Wesley, 1990.
53. 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(5):295–310, 2004.

54. A. J. C. van Gemund. Symbolic Performance Modeling of Parallel Systems. *IEEE Trans. on Parallel and Distributed Systems*, 14(2):154–165, 2003.
55. D. Bertsimas and D. Gamarnik. Asymptotically optimal algorithm for job shop scheduling and packet routing. *Journal of Algorithms*, 33(2):296–318, 1999.
56. 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* 19^{th} *International Parallel and Distributed Processing Symposium (IPDPS 05) (IPDPS'05)*, April 2005.
57. O. Beaumont, A. Legrand, L. Marchal, and Y. Robert. Steady-State Scheduling on Heterogeneous Clusters: Why and How? In *Proc. of* 18^{th} *International Parallel and Distributed Processing Symposium (IPDPS 04) (IPDPS'04)*, April 2004.
58. 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(4):319–330, April 2004.
59. J. Schopf. Structural prediction models for high-performance distributed applications. In *Proc. of the Cluster Computing Conference (CCC'97)*, Atlanta, USA, March 1997.
60. M. Dìaz, B. Rubio, E. Soler, and J. M. Troya. SBASCO: Skeleton-based Scientific Components. In *Proc. of* 12^{th} *Euromicro Conference on Parallel, Distributed, and Network-Based Processing (PDP'04)*, A Coruña, Spain, February 2004.
61. 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.
62. W. Smith, I. T. Foster, and V. E. Taylor. Predicting Application Run Times Using Historical Information. In *Proc. of the Workshop on Job Scheduling Strategies for Parallel Processing (IPPS/SPDP 98)*, pages 122–142, London, UK, 1998. Springer-Verlag.
63. F. Vraalsen, R. A. Aydt, C. L. Mendes, and D. A. Reed. Performance Contracts: Predicting and Monitoring Grid Application Behavior˙In *Proc. of* 2^{nd} *International Workshop on Grid Computing (GRID 01)*, pages 154–165, London, UK, 2001. Springer-Verlag.
64. L. J. Senger, M. J. Santana, and R. H. C. Santana. Using Runtime Measurements and Historical Traces for Acquiring Knowledge in Parallel Applications. In M. Bubak, G. D. van Albada, P. M. Sloot, and J. J. Dongarra, editors, *Proc. of the 2004 International Conference on Computational Science (ICCS 04)*, volume 3036 of *Lecture Notes in Computer Science*, pages 661–665, Kraków, Poland, June 2004. Springer Verlag.
65. 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.
66. N. Dumitrascu, S. Murphy, and L. Murphy. A Methodology for Predicting the Performance of Component-Based Applications. In *Proc. of* 8^{th} *International Workshop on Component-Oriented Programming (WCOP 03)*, Darmstadt, Germany, July 2003.
67. 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* 18^{th} *International Parallel and Distributed Processing Symposium (IPDPS 04)*, Santa Fé, USA, April 2004.
68. 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.
69. B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1997.
70. A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making Components Contract Aware. *Computer*, 32(7):38–45, 1999.
71. D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim. Jass - Java with Assertions. In *Proc. of* 13^{th} *Conference on Computer Aided Verification (CAV 01)*, 2001.
72. Object Management Group. The Object Constraint Language Specification. `http://www.omg.org/docs/formal/06-05-01.pdf`, May 2006.
73. M. Barnett, W. Grieskamp, C. Kerer, W. Schulte, C. Szyperski, N. Tillmann, and A. Watson. Serious Specification for Composing Components. In I. Crnkovic, H. Schmidt, J. Stafford, and K. Wallnau, editors, *Proc. of* 6^{th} *ICSE Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction*, May 2003.

74. S. Frolund and J. Koistinen. QML: A Language for Quality of Service Specification. Technical Report HPL-98-10, Hewlett-Packard Laboratories, Palo Alto, CA, USA, 1998.
75. J. O. Aagedal. *Quality of Service Support in Development of Distributed Systems*. PhD thesis, University of Oslo, 2001.
76. S. Röttger and S. Zschaler. CQML$^+$: Enhancements to CQML. In *Proc. of* 1^{st} *International Workshop on Quality of Service in Component-based Software Engineering*, pages 43–56, 2003.
77. M. Barnett and W. Schulte. Runtime verification of .NET contracts. *Journal of System Software*, 65(3):199–208, 2003.
78. E. Teiniker, R. Lechner, G. Schmoelzer, C. Kreiner, Z. Kovacs, and R. Weiss. Towards a Contract Aware CORBA Component Container. In *Proc. of* 29^{th} *Annual International Computer Software and Applications Conference (COMPSAC 05)*, pages 545–550, Washington, DC, USA, 2005. IEEE Computer Society.
79. P. Caruso, G. Laccetti, and M. Lapegna. A Performance Contract System in a Grid Enabling, Component Based Programming Environment. In P. M. A. Sloot, A. G. Hoekstra, T. Priol, A. Reinefeld, and M. Bubak, editors, *Proc. of Advances in Grid Computing, European Grid Conference (EGC 2005)*, volume 3470 of *Lecture Notes in Computer Science*, pages 982–992, Amsterdam, The Netherlands, February 2005. Springer Verlag.
80. N. H. Kapadia, J. A. B. Fortes, and C. E. Brodley. Predictive Application-Performance Modeling in a Computational Grid Environment. In *Proc. of* 8^{th} *IEEE International Symposium on High Performance Distributed Computing (HPDC 99)*, pages 47–54, Washington, DC, USA, 1999. IEEE Computer Society.
81. B. Spitznagel and D. Garlan. Architecture-Based Performance Analysis. In Y. Deng and M. Gerken, editors, *Proc. of* 10^{th} *International Conference on Software Engineering and Knowledge Engineering (SEKE 98)*, pages 146–151, 1998.
82. I. Foster and C. Kesselman. Computational Grids. In I. Foster and C. Kesselman, editors, *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan Kaufmann Publishers, 1998.
83. FAFNER: Factoring via network enabled recursion. Web Page, Syracuse University, `http://www.npac.syr.edu/factoring.html`.
84. I. Foster, J. Geisler, W. Nickless, W. Smith, and S. Tuecke. Software Infrastructure for the I-WAY High Performance Distributed Computing Experiment. In *Proc.* 5^{th} *IEEE Symposium on High Performance Distributed Computing*, pages 562–571, 1997.
85. R. Aiken, M. Carey, B. Carpenter, I. Foster, C. Lynch, J. Mambretti, R. Moore, J. Strasnner, and B. Teitelbaum. Network Policy and Services: A Report of a Workshop on Middleware. IETF, RFC 2768, `http://www.ietf.org/rfc/rfc2768.txt`, February 2000.
86. The Globus Toolkit. Web Page, Globus Alliance, `http://www.globus.org/toolkit/`.
87. UNICORE: Uniform Interface to Computing Resources. Web Page, `http://www.unicore.eu`.
88. LEGION: Worldwide Virtual Computer. Web Page, University of Virginia, `http://legion.virginia.edu/`.
89. I. Foster. What is the Grid? A Three Point ChecklistWhat is the Grid? A Three Point Checklist. *GRIDtoday*, 1(6), July 2002.
90. I. Foster and C. Kesselman. Concepts and Architecture. In I. Foster and C. Kesselman, editors, *The Grid 2: Blueprint for a Future Computing Infrastructure*, chapter 4, pages 37–64. Morgan Kaufmann Publishers, 1^{st} edition, 2004.
91. I. Foster and C. Kesselman. The Grid in a Nutshell. In J. Nabrzyski, J. M. Schopf, and J. Weglarz, editors, *Grid Resource Management: state of the art and future trends*, chapter 1. Kluwer Academic Publishers, Norwell, MA, USA, 2004.
92. N. Tonellotto, P. Wieder, and R. Yahyapour. A Proposal for a Generic Grid Scheduling Architecture. In S. Gorlatch and M. Danelutto, editors, *Integrated Research in GRID Computing*, CoreGRID series, pages 227–239. Springer Verlag, 2007.
93. U. Schwiegelshohn and R. Yahyapour. Attributes for Communication between Scheduling Instances. In J. Nabrzyski, J. M. Schopf, and J. Weglarz, editors, *Grid Resource Management: state of the art and future trends*, chapter 4. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

94. J. M. Schopf. Ten Actions When Grid Scheduling. In J. Nabrzyski, J. M. Schopf, and J. Weglarz, editors, *Grid Resource Management: state of the art and future trends*, chapter 2. Kluwer Academic Publishers, Norwell, MA, USA, 2004.
95. T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Trans. on Software Engineering*, 14(2):141–154, 1988.
96. F. Dong and S. G. Akl. Scheduling Algorithms for Grid Computing: State of the Art and Open Problems. Technical Report 2006-504, School of Computing, Queen's University, Kingston, Ontario, Canada, January 2006.
97. T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 61(6):810–837, 2001.
98. H. Casanova, D. Zagorodnov, F. Berman, and A. Legrand. Heuristics for Scheduling Parameter Sweep Applications in Grid Environments. In *Proc. of* 9^{th} *Heterogeneous Computing Workshop (HCW00)*, page 349, Washington, DC, USA, 2000. IEEE Computer Society.
99. H. Chen and M. Maheswaran. Distributed Dynamic Scheduling of Composite Tasks on Grid Computing Systems. In *Proc. of* 16^{th} *International Parallel and Distributed Processing Symposium (IPDPS02)*, page 119, Washington, DC, USA, 2002. IEEE Computer Society.
100. Y. Zhu. A Survey on Grid Scheduling Systems. Technical report, Computer Science Department, Hong Kong University of Science and Technology, Hong Kong, China, 2003.
101. F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov. Adaptive Computing on the Grid Using AppLeS. *IEEE Trans. on Parallel and Distributed Systems*, 14(4):369–382, 2003.
102. D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the Condor experience. *Concurrency and Computation: Practice and Experience*, 17(2–4):323–356, February 2005.
103. OASIS. Reference Model for Service Oriented Architecture (SOA-RM). `http://www.oasis-open.org/committees/documents.php`, August 2006.
104. AA. VV. *The Emergence of Grid and Service-Oriented IT: An Industry Vision for Business Success*. Tabor Communications, Inc., 2006.
105. X. He, X. Sun, and G. von Laszewski. Qos guided min-min heuristic for grid task scheduling. *Journal of Computer Science and Technology*, 18(4):442–451, 2003.
106. M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund. Dynamic Matching and Scheduling of a Class of Independent Tasks onto Heterogeneous Computing Systems. In *Proc. of* 8^{th} *Heterogeneous Computing Workshop (HCW 99)*, Washington, DC, USA, 1999. IEEE Computer Society.
107. O. Sinnen and L. Sousa. A Classification of Graph Theoretic Models for Parallel Computing. Technical Report RT/005/99, INESC-ID, Instituto Superior Técnico, Technical University of Lisbon, Portugal, May 1999.
108. H. S. Stone. Multiprocessor Scheduling with the Aid of Network Flow Algorithms. *IEEE Trans. on Software Engineering*, SE-3(1):85–93, 1977.
109. M. G. Norman and P. Thanisch. Models of machines and computation for mapping in multicomputers. *ACM Computing Surveys*, 25(3):263–302, September 1993.
110. S. H. Bokhari. A Shortest Tree Algorithm for Optimal Assignments Across Space and Time in a Distributed Processor System. *IEEE Trans. on Software Engineering*, SE-7(6):583–589, 1981.
111. D. Fernández-Baca. Allocating Modules to Processors in a Distributed System. *IEEE Trans. on Software Engineering*, 15(11):1427–1436, 1989.
112. M. Kafil and I. Ahmad. Optimal Task Assignment in Heterogeneous Distributed Computing Systems. *IEEE Concurrency*, 6(3):42–51, 1998.
113. M. K. Dhodhi, I. Ahmad, A. Yatama, and I. Ahmad. An integrated technique for task matching and scheduling onto distributed heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 62(9):1338–1361, 2002.
114. Y.-C. Ma, T.-F. Chen, and C.-P. Chung. Branch-and-bound task allocation with task clustering-based pruning. *Journal of Parallel and Distributed Computing*, 64(11):1223–1240, 2004.

115. B. Ucar, C. Aykanat, K. Kaya, and M. Ikinci. Task assignment in heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 66(1):32–46, 2006.
116. J. Licklider and W. Clark. On-Line Man Computer Communication, August 1962.
117. L. Kleinrock. UCLA to be first station in nationwide computer network. UCLA Office of Public Information, `http://www.lk.cs.ucla.edu/LK/Bib/REPORT/press.html`, July 1969.
118. R. Baraglia, M. Danelutto, T. Fagni, D. Laforenza, S. Orlando, A. Paccosi, N. Tonellotto, M. Vanneschi, and C. Zoccolo. HPC Application Execution on Grids. In V. Getov, D. Laforenza, and A. Reinefeld, editors, *Future Generation Grids*, CoreGRID series, pages 263–282. Springer Verlag, November 2005.
119. D. Adami, S. Giordano, M. Repeti, M. Coppola, D. Laforenza, and N. Tonellotto. Design and Implementation of a Grid Network-Aware Resource Broker. In T. Fahringer, editor, *Proceedings of IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN2006)*. IASTED, ACTA press, February 2006.
120. M. Danelutto, D. Laforenza, N. Tonellotto, M. Vanneschi, and C. Zoccolo. A Performance Model for Stream-based Computations. In P. D'Ambra and M. M. Guarracino, editors, *Proceedings of the* 15^{th} *Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP2007)*, pages 91–96, Napoli, Italy, February 2007. IEEE Computer Society Press.
121. N. Tonellotto and C. Zoccolo. Characterization of the performance of ASSIST programs. Technical Report TR-0007, Institute on Programming Model & Institute on Resource Management and Scheduling, CoreGRID - Network of Excellence, June 2005.
122. M. Danelutto, D. Laforenza, N. Tonellotto, M. Vanneschi, and C. Zoccolo. A Performance Model for Stream-based Computations. Submitted to *International Journal in Computer Science and Engineering*, Jul 2007.
123. C.-T. Chen. *Linear System Theory and Design*. Oxford University Press, 3^{rd} edition, 1998.
124. M. Coppola, M. Danelutto, D. Laforenza, N. Tonellotto, M. Vanneschi, and C. Zoccolo. Managing user expectations with component performance contracts. In O. Rana, P. Wieder, W. Ziegler, and R. Yahyapour, editors, *Proceedings of the CoreGRID Workshop on Usage of Service Level Agreements in Grids*, Austin, Texas, USA, September 2007. CoreGRID, IST.
125. A. Litke, A. Panagakis, A. Doulamis, N. Doulamis, T. Varvarigou, and E. Varvarigos. An Advanced Architecture for a Commercial Grid Infrastructure. In *Grid Computing*, volume 3165 of *Lecture Notes in Computer Science*, pages 32–41, Berlin, Germany, 2004. Springer Verlag.
126. R. Baraglia, R. Ferrini, N. Tonellotto, D. Adami, S. Giordano, and R. Yahyapour. A Study on Network Resources Management. In S. Gorlatch, M. Bubak, and T. Priol, editors, *Integrated Research in GRID Computing*, pages 213–224. Academic Computer Centre CYFRONET AGH, Kraków, Poland, October 2006.
127. R. Baraglia, R. Ferrini, N. Tonellotto, L. Ricci, and R. Yahyapour. A Launch-time Scheduling Heuristics for Parallel Applications on Wide Area Grids. *Journal of Grid Computing,* Online First, 2007.
128. H.-U. Heiss. Mapping Tasks onto Processors at Run-time. In E. Gelenbe, U. Halici, and N. Yalabik, editors, *Proc. of* 7^{th} *International Symposium on Computer and Information Sciences (ISCIS VII)*, pages 515–518, Antalya, Turkey, November 1992.
129. R. Wolski, N. T. Spring, and J. Hayes. The Network Weather Service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computing Systems*, 15(5-6):757–768, 1999.
130. M. den Burger, T. Kielmann, and H. Bal. TOPOMON: A monitoring tool for grid network topology. In *Proc. of the International Conference on Computational Science*, Berlin, Germany, 2002. Springer Verlag.
131. E. Demaine and N. Immorlica. Correlation Clustering with Partial Information. In *Proc. of* 6^{th} *International Workshop on Approximation Algorithms for Combinatorial Optimization Problems*, Berlin, Germany, 2003. Springer Verlag.
132. R. Adams and L. Bischof. Seeded Region Growing. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 16(6):641–647, 1994.

133. M. Doar. A Better Model for Generating Test Networks. In *Proceedings of Globecom '96*, 1996.
134. R. Baraglia, R. Ferrini, N. Tonellotto, L. Ricci, and R. Yahyapour. QoS-constrained List Scheduling Heuristics for Parallel Applications on Grids. Technical Report TR-0093, Institute on Resource Management and Scheduling, CoreGRID - Network of Excellence, August 2007.
135. R. Baraglia, R. Ferrini, L. Ricci, N. Tonellotto, and R. Yahyapour. QoS-constrained List Scheduling Heuristics for Parallel Applications on Grids. In J. Bourgeois and D. El Baz, editors, *Proceedings of the* 16^{th} *Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP2008)*, Toulose, France, February 2008. IEEE Computer Society Press.