UNIVERSITY OF UDINE - ITALY

Department of Mathematics and Computer Science

Ph.D. Thesis

DAG SCHEDULING FOR GRID COMPUTING SYSTEMS

Supervisors:
Prof. ALESSANDRO DE ANGELIS

Candidate:
ALBERTO FORTI

Doctorate of Philosophy in Computer Science

XVIII cycle

AY 2005/2006

Abstract

Today's parallel and distributed systems are changing in their organization and the concept of Grid computing, a set of dynamic and heterogeneous resources connected via Internet and shared by many and different users, is nowadays becoming a reality. A large number of scheduling heuristics for parallel applications described by directed acyclic graphs (DAGs) have been presented in the literature, but most of them assume a homogeneous system with a homogeneous network, i.e. a message is transmitted with the same speed on all the links. In a Grid environment this assumption cannot be done. In this thesis we tackle the problem of scheduling parallel applications described by directed acyclic graphs (DAGs) in a Grid computing system.

Contents

Introduction

The subject of this thesis is the scheduling of parallel applications described by directed acyclic graphs (DAGs) in Grid computing systems. Basically, Grid is a geographical distributed system. Distributed computing scaled to a global level with the maturation of the Internet in the 1990s. Advances in networking technologies will soon make it possible to use the global information infrastructure in a qualitatively different way, as a computational as well as an information resource. The last decade has seen a substantial increase in commodity computer and network performance, as a result of faster hardware, low costs and more sophisticated software. Nevertheless, there are problems in the fields of science, engineering, and business, which cannot be effectively dealt with using the current generation of supercomputers. Usually these problems are computational and data intensive and consequently need heterogeneous resources not available in a single organization. The ubiquity of the Internet and the availability of high-speed networks leads to the possibility of using wide-area distributed computers for solving large-scale problems. Such an approach to network computing is known by several names: metacomputing, scalable computing, global computing, Internet computing, Peer-to-Peer (P2P) and Grid computing.

Grids enable the sharing, selection, and aggregation of a wide variety of resources including supercomputers, storage systems, data sources, and specialized devices that are geographically distributed and owned by different organizations. The Grid allows users to solve larger-scale problems by pooling together resources that could not be coupled easily before.

The concept of Grid computing started as a project to link geographically dispersed supercomputers, but now it has grown far beyond its original intent. Due to the rapid growth of the Internet and Web, there has been a growing interest in Web-based distributed computing, and many projects have been started and aim to exploit the Web as an infrastructure for running coarse-grained distributed and parallel applications. In fact, like any other growing idea, the Grid has evolved passing through different phases. Nowadays we are in the third Grid generation [106]. This generation defines the Grid as a service oriented architecture based on web services. In this context, the Web has the capability to act as a platform for parallel and collaborative work as well as a key technology to create a pervasive and ubiquitous Grid-based infrastructure.

The design of Grid as a service oriented architecture have led to a growing interest in workflows by the Grid community. The reason is that the "software as a service" approach results in a componentized view of software applications and workflow can naturally be used as a component composition mechanism. Traditionally, the main applications of workflows have been in the automation of administrative and production processes, especially within businesses and large organizations. The expansion of workflow towards middleware is realized within the Web service initiative. Grid workflows are an emerging research field in the Grid community and there is an ongoing effort to define a standard meaning of workflow for the Grid. Actually, the most common Grid workflow can be modelled as simple Task (Directed Acyclic) Graphs (DAGs), where the order of execution of tasks (modelled as nodes) is determined by dependencies (in turn modelled as directed arcs). Each DAG node represents the execution of a component, characterized by a set of attributes such as an estimate of its cost and possible requirements on the target execution platform, while DAG directed edges represent data dependencies between specific application components. Data dependencies will be usually constituted by large files written by a component and required for the execution of one or more other components of the application. Two types of DAGs can be distinguished: coarse grained DAGs, in which the computation is dominant with respect to communication, and fine grained DAGs, in which the communication is dominant with respect to computation.

One of the most active areas of research in the Grid community is scheduling. Scheduling in Grid means taking decisions involving resources distributed over multiple administrative domains. The main difference between a Grid scheduler and a local scheduler is that the former does not own the resources at a local site and has no control over them. In particular it doesn't know if other users are sending jobs to the resources that it is considering. Grid is a dynamic environment and resources come and go, therefore a scheduler has to be able to discover and monitor the resources. In general, Grid schedulers get information from a general Grid Information System (GIS) that in turn gathers information from individual local resources. The Globus Monitoring and Discovery Service [26] is an example of Grid service that allows the monitoring and the discovery of the resources. Another example is the Network Weather System [130] that is a distributed monitoring system designed to track and forecast dynamic resource conditions, e.g. it allows a user or a program (such as a scheduler) to request information (latency, bandwidth, load, estimates, etc) for entities corresponding to network links connecting specified endpoints, retrieve the fraction of CPU available to a newly started process, retrieve the amount of memory that is currently unused in a remote host, etc.

The problem of scheduling DAGs in a heterogeneous environment is NP-complete. To tackle the problem, simplifying assumptions have been made regarding the task graph structure representing the program and the model for the parallel processor system. However, it is NP-complete even in two simple cases: (1) scheduling unit-time tasks to an arbitrary number of processors [58], (2) scheduling one or two time unit tasks to two processors [22]. Many DAG scheduling algorithms can be found in the literature. Usually they do not impose constraints on the graph structure but many of them assume a homogeneous computing system. The reason is that they were designed to schedule parallel applications in clusters. For the same reason the algorithms that consider heterogeneous systems assume a homogeneous interconnection network, i.e. all the links have the same latency and bandwidth and, therefore, messages are transmitted with the same speed on all the links. Very few works do not impose any constraint on the networked computing system.

The DAG scheduling problem is usually identified by the combination of two phases: *matching* which assigns tasks to machines and *scheduling* which defines the execution order of the tasks assigned to each machine. The overall problem of matching and scheduling is referred to as *mapping*.

Due to the intractability of the general scheduling problem, many heuristics have been suggested to tackle it under more generic situations. A heuristic produces an answer in less than exponential time but does not guarantee an optimal solution. The most popular approach to the design of scheduling algorithms is the *list scheduling* technique. Tasks are associated to priorities and a selection phase chooses the one with higher priority. The selected task is then mapped to an appropriate resource. In order to find a good schedule, the scheduler must take care of these two aspects:

- Map a task to a resource that allows to complete the execution of that task as soon as possible.

- Minimize data transfer times.

Obviously it is impossible to optimize both aspects at the same time, a tradeoff must be found. In particular, minimization of the data transfer times can be accomplished in two ways: by mapping the communicating tasks onto two resources close to each other or with the zeroing of the communication time by mapping the two tasks onto the same resource. Usually, the tradeoff point is between parallelization and sequentialization. To sequentialize means mapping parallel tasks onto the same resource, sequentialization allows to zero incoming or outgoing edges connecting the two tasks to a common parent or child. What emerges from the literature is that algorithms based on *Critical Path* (CP) heuristics are the ones giving, on average, the best results in terms of quality of the schedule produced, i.e. reduction of the completion time of the entire workflow. The CP is the weight of the longest path of the DAG and provides an upper bound on the schedule length. In order to keep track of the CP during the scheduling some algorithms consider nodes belonging to the *Dominant Sequence* (DS), that is the CP of the scheduled DAG. Anyway, even these dynamic heuristics can get trapped in a locally optimal decision, leading to a non-optimal global solution. This means that scheduling at each step a DS node may not be the correct choice.

This thesis aims at studying algorithms for scheduling DAGs in a Grid computing system. We first have analyzed the problem of scheduling in the Grid and then we have studied how the DAG scheduling algorithms found in the literature work. The final goal of this thesis is to propose a novel scheduling algorithm to address the problem of scheduling DAG described parallel applications in Grid. A modification of the well known DSC [135] algorithm is also presented. We have conducted extensive simulation tests in order to compare the results of the two proposed algorithms with other reference algorithms.

Thesis outline

This thesis is divided into six chapters, which are organized as follows.

In order to make the reader familiar with Grid computing, in chapter 1 we present an introduction to Grid, together with a case study of the migration to Grid of the astroparticle experiment (in which we have participated) MAGIC.

In chapter 2 we describe the evolution of Grid together with the technologies that enables this new paradigm of computation.

In chapter 3 we review Grid scheduling issues and challenges. Then, we consider how workflows have been characterized by the Grid community and we review some Grid scheduling systems developed in the last years.

In chapter 4 we explain in detail the DAG scheduling problem and review the actual state of the art of scheduling algorithms.

In chapter 5 we propose two DAG scheduling algorithms designed to work in Grid. The first one is called CCF (Cluster ready Children First) and the second one, the DSC_VAR, is a variant of the famous DSC. Some properties of the proposed CCF algorithm are identified and analyzed.

Finally, chapter 6 presents experimental results obtained with the simulation of the proposed algorithms CCF and DSC_VAR.

1 Introduction to Grid

Next generation scientific exploration requires computing power and storage that no single institution alone is able to afford. Additionally, easy access to distributed data is required to improve the sharing of results by scientific communities spread around the world. The proposed solution to these challenges is to enable different institutions, working in the same scientific field, to put their computing, storage and data resources together in order to achieve the required performance and scale. Grid is a type of parallel and distributed system that enables the sharing, selection, and aggregation of services of heterogeneous resources distributed across “multiple” administrative domains based on their availability, capability, performance, cost, and users' quality-of-service requirements. As Network performance has outpaced computational power and storage capacity, this new paradigm has evolved to enable the sharing and coordinated use of geographically distributed resources. This chapter presents an introduction to Grid giving focus to the main requirements and challenges that must be addressed in setting up this new paradigm of distributed computing.

1.1 What is Grid

The ancestor of the Grid is Metacomputing. This term was coined in the early eighties. The idea of Metacomputing was to interconnect a collection of computers held together by state-of-the-art technology and ”balanced” so that, to the individual user, it looks and acts like a single computer. The constituent parts of the resulting “metacomputer” could be housed locally, or distributed between buildings, even continents. One of the first infrastructures in this area, named Information Wide Area Year (I-WAY) [41], was demonstrated at Supercomputing 1995. This project strongly influenced the subsequent Grid computing activities. In fact one of the researchers who lead the project I-WAY was Ian Foster who along with Carl Kesselman published in 1997 a paper [42] that clearly links the Globus Toolkit, which is currently the heart of many Grid projects, to Metacomputing.

Foster and Kesselman published in 1998 the book “The Grid: Blueprint for a New Computing Infrastructure”[43], which is considered the “Grid bible”. They defined the Grid as follows: *A computational Grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end* *computational capabilities.* In fact one of the main ideas of the Grid, which also explains the origin of the word itself, was to make computational resources available like electricity. One remarkable fact of the electric power grid infrastructure is that when we plug an appliance into it we do not care where the generators are located and how they are wired. We are only interested in getting the electric power, and that's all! Unfortunately, in practice, the similarities between the electric power grid and the computational Grid are very few.

According to a Foster's check list the minimum properties of a Grid system are the following [47]:

- A Grid coordinates resources that are not subject to centralized control (e.g. resources owned by different companies or under the control of different administrative units) and at the same time addresses the issues of security, policy, payment, membership, and so forth that arise in these settings.
- A Grid must use standard, open, general-purpose protocols and interfaces. These protocols address fundamental issues such as authentication, authorization, resource discovery, and resource access.
- A Grid delivers nontrivial quality service, i.e. it is able to meet complex user demands (e.g. response time, throughput, availability, security, etc.).

This checklist gives focus to what is the key concept of Grid computing: the ability to negotiate resource-sharing arrangements among a set of participating parties (providers and consumers). The "sharing" refers not only to file exchange but also to direct access to computers, software, data, and other resources. This sharing have to be highly controlled 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. A set of individuals and/or institutions defined by such sharing rules forms a **virtual organization**.

Basically, the vision that is now becoming reality is as follows:

- The user *submits* his request through a Graphical User Interface (GUI) just specifying high level requirements (the kind of application he wants to use, the operating system,...) and possibly providing input data.
- The Grid *finds* and allocates suitable resources (computing systems, storage facilities, etc.) to satisfy the user's request.
- The Grid monitors request processing.
- The Grid notifies the user when the results are available allowing their retrieval.

Grid can be seen as the latest and most complete evolution of more familiar developments such as distributed computing, the Web, peer-to-peer computing and virtualization technologies:

- Like the Web, Grid computing keeps complexity hidden: multiple users enjoy a single, unified experience.

- Unlike the Web, which mainly enables communication, Grid computing enables full collaboration toward common business goals.
- Like peer-to-peer, Grid computing allows users to share files.
- Unlike peer-to-peer, Grid computing allows many-to-many sharing not only files but other resources as well.
- Like clusters and distributed computing, Grids bring computing resources together.
- Unlike clusters and distributed computing, which need physical proximity and operating homogeneity, Grids can be geographically distributed and heterogeneous.
- Like virtualization technologies, Grid computing enables the virtualization of IT resources.
- Unlike virtualization technologies, which virtualize a single system, Grid computing enables the virtualization of vast and disparate IT resources.

1.1.1 Grid challenges

There are many challenges that must be addressed in order to build a working Grid environment. The following list shows the main requirements a Grid should satisfy, or equivalently the main services a Grid should make available:

- **Information services**: Concerns information about the resources available on the Grid. It includes the set of available resources, hardware specifications as well dynamic information like load and forecasts. These information should be automatically maintained and kept up to date.
- **Resource brokering**: Grid users should submit their requests to a resource broker specifying their high level requirements. The Resource Broker should be able to find and allocate suitable resources by querying information services.
- **Uniform access to resources**: all the resources of the same kind (computing elements, storage elements, etc.) should be accessed in a uniform way, no matter which technologies or standards they are based on. Middleware installed on each single machine is a way for hiding heterogeneity and for providing uniform interfaces.
- **Security**: Security mechanisms are needed in order to enable system administrators to enforce access rules for all the resources made available on the Grid. This point is strongly related with the concept of virtual organization. A number of issues must be addressed inside the security context [46]:
  - *Single sign-on.* A single computation may entail access to many resources, but a user should be able to authenticate once and then assign to the computation the right to operate on his or her behalf.

  - *Mapping to local security mechanisms.* Different sites may use different local security solutions. A Grid security infrastructure needs to map to these local solutions at each site, so that local operations can proceed with appropriate privileges.
  - *Delegation.* A computation that spans many resources creates sub-computations that may themselves generate requests to other resources and services, perhaps creating additional subcomputations, and so on. Authentication operations are involved at each stage.
  - *Community authorization and policy.* In a large community, the policies that govern who can use which resources for what purpose cannot be based directly on individual identity. It is infeasible for each resource to keep track of community membership and privileges. Instead, resources (and users) need to be able to express policies in terms of other criteria, such as group membership, which can be identified with a cryptographic credential issued by a trusted third party.

- **Job scheduling**: Jobs submitted by the users should be effectively and efficiently scheduled.
- **Data Access**: Grid users should be able to access distributed data in a uniform way.
- **Data replication**: Grids should allow automatic file replica creation in order to move data closer to the user or to the computing facilities that will process them. It is also a way to increase the fault-tolerance of the system.

1.2 Grid Applications

Currently there is a big effort in helping applications to migrate to Grid. An example is the EGEE (Enabling Grids for E-sciencE) project [34]. The project aims to provide researchers in academia and industry with access to major computing resources, independently of their geographic location. The EGEE project will also focus on attracting a wide range of new users to the Grid. Two pilot application domains have been selected to guide the implementation and certify the performance and functionality of the evolving infrastructure. One is the Large Hadron Collider Computing Grid, supporting HEP (High Energy Physic) physics experiments, and the other is Biomedical Grids, where several communities are facing equally daunting challenges to cope with the flood of bioinformatics and healthcare data.

Just to give an example, in the following section we briefly describe the recent migration to Grid of one astroparticle physics experiment: the MAGIC telescope.

1.2.1 Case study: the MAGIC telescope

The MAGIC (Major Atmospheric Gamma Imaging Cerenkov telescope) telescope has been designed to search the sky to discover or observe high energy γ-rays sources and address a large number of physics questions [87]. Located at the Instituto Astrophysico de Canarias on the island La Palma, Spain, at 28° N and 18° W, at altitude 2300m asl, it is the largest γ-ray telescope in the world. MAGIC is operating since October 2003, data are taken regularly since February 2004 and signals from Crab and Markarian 421 was seen.

The main characteristics of the telescope are summarized below:

- A $17m$ diameter (f/d=1) tessellated mirror mounted on an extremely light carbon-fiber frame (< 10 tons), with active mirror control. The reflecting surface of mirrors is $240m^2$; reflectivity is larger than 85% (300 - 650nm).
- Elaborate computer-driven control mechanism.
- Fast slewing capability (the telescope moves 180° in both axes in $22s$).
- A high-efficiency, high-resolution camera composed by an array of 577 fast photomultipliers (PMTs), with a 3.9° field of view.
- Digitalization of the analogue signals performed by 300 MHz FlashADCs and a high data acquisition rate of up to 1 KHz.
- MAGIC is the lowest threshold ($\approx$ 30 GeV) IACT operating in the world.

γ-ray observation in the energy range from a few tenths of GeV upwards, in overlap with satellite observations and with substantial improvements in sensitivity, energy and angular resolution, leads to search behind the physics that has been predicted and new avenues will open. Understanding of AGNs, GRBs, SNRs, Pulsars, diffuse photon background, unidentified EGRET sources, particle physics, darkmatter, quantum gravity and cosmological γ-ray horizon are some of the physics goals that can be addressed with the MAGIC telescope.

Benefits of Grid computing for MAGIC

The collaborators of the MAGIC telescope are mainly spread over Europe, 18 institutions from 9 countries, with the main contributors (90% of the total) located in Germany (Max-Planck-Institute for Physics, Munich and University of Wuerzburg), Spain (Barcelona and Madrid), Italy (INFN and Universities of Padova, Udine and Siena).

The geographical distribution of the resources makes the management of the experiment harder. This is a typical situation for which Grid computing can be of great help, because it allows researchers to access all the resources in a uniform, transparent and easy way. The telescope is in operation during moonless nights. The average amount of raw FADC data recorded is about 500-600 GB/night. Additional data from the telescope control system or information from a weather station are also recorded. All these information have to be taken into account in the data analysis.

The MAGIC community can leverage from Grid facilities in areas like file sharing, Monte Carlo data production and analysis [40]. In a Grid scenario the system can be accessed through a web browser based interface with single sign-on authentication

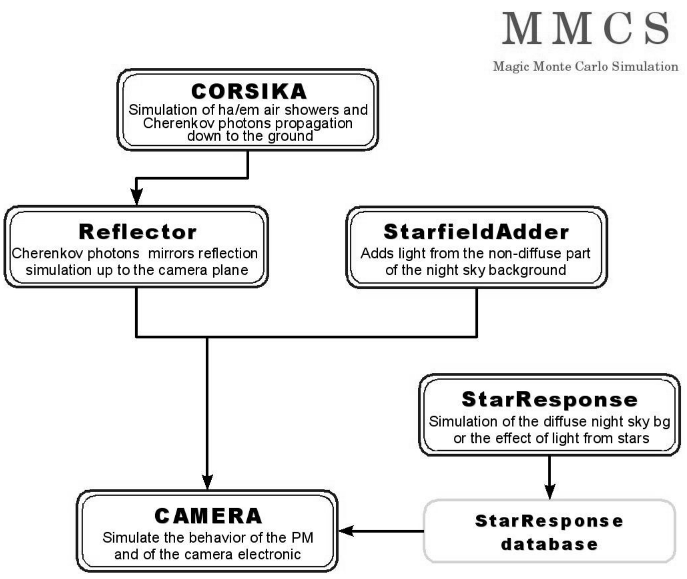

MMCS
Magic Monte Carlo Simulation
CORSIKA
Simulation of ha/em air showers and Cherenkov photons propagation down to the ground
Reflector
Cherenkov photons mirrors reflection simulation up to the camera plane
StarfieldAdder
Adds light from the non-diffuse part of the night sky background
StarResponse
Simulation of the diffuse night sky bg or the effect of light from stars
CAMERA
Simulate the behavior of the PM and of the camera electronic
StarResponse database

Figure 1.1: MAGIC Monte Carlo Simulation workflow

method. We can briefly summarize the main benefits given by the adoption of Grid technology for the MAGIC experiment:

- Presently, users analyzing data must know where to find the required files and explicitly download them. In a Grid perspective, instead, users don't care about data location and files replication policies improve access time and fault tolerance.
- Grid workflow tools can manage the MAGIC Monte Carlo simulation. The resources from all the members of the MAGIC community can be put together and exploited by the Grid. Easy access to data production for every user, or accordingly to the virtual organization (VO) policies.
- Analysis tools can be installed on the Grid. They are thus shared and available for all the users (no need for single installations). Moreover, they can exploit the facilities of a distributed system.

MMCS

The MAGIC Monte Carlo Simulation workflow is a series of programs which simulate the properties of different physics processes and detector parts (figure 1.1):

- *CORSIKA*: air shower and hadronic background simulation. The output contains information about the particles and the Cherenkov photons reaching the ground around the telescope.

- *Reflector*: simulates the propagation of Cherenkov photons through the atmosphere and their reflection in the mirror up to the camera plane. The input for the Reflector program is the output of CORSIKA.
- *StarfieldAdder*: simulation of the field of view. It adds light from the non-diffuse part of the night sky background, or the effect of light from stars, to images taken by the telescope.
- *StarResponse*: simulation of the night sky background (NSB) response.
- *Camera*: simulate the behavior of the photomultipliers and of the electronic of the MAGIC camera. It also allows to introduce the NSB (optionally), from the stars and/or the diffuse NSB.

Architecture

Figure 1.2 shows the computing centers that make available the main resources forming the backbone of the Grid system for MAGIC. These centers are: GRIDKA (Germany), CNAF (Italy) and PIC (Spain). The system [70] will be based on the middleware from the European Data Grid project [121], which is using the Globus toolkit [42] as the underlying software. The data flow starts in the island of La Palma and arrives in all the other centers passing through the PIC institute in Barcellona. The two services of the system are the MAGIC Monte Carlo Simulation (MMCS) and the MAGIC Analysis and Reconstruction Software (MARS). These services will run at all sites. Each of the two services will have its own scheduler (Resource Broker) running at CNAF or PIC. The schedulers will send the jobs to the different sites. The produced Monte Carlo data will be distributed as will the incoming real data from the telescope. The distribution and replication of the data will be based on the replica location service of the EDG project. The system will be accessed via a portal running at GRIDKA. The local computers from a MAGIC partner site can connect to the closest of the backbone nodes to contribute with their local computing resources.

1.3 Summary

Increased network bandwidth, more powerful computers, and the acceptance of the Internet have driven the on-going demand for new and better ways to compute. At the heart of Grid Computing is an infrastructure that provides dependable, consistent, pervasive and inexpensive access to computational capabilities. By pooling federated assets into a virtual system, a grid provides a single point of access to powerful distributed resources. With a Grid, networked resources, e.g. desktops, servers, storage, databases, even scientific instruments, can be combined to deploy massive computing power wherever and whenever it is needed most. Users can find resources quickly, use them efficiently, and scale them seamlessly.

A Grid can be seen as the latest and most complete evolution f more familiar developments such as distributed computing, the Web, peer-to-peer computing and

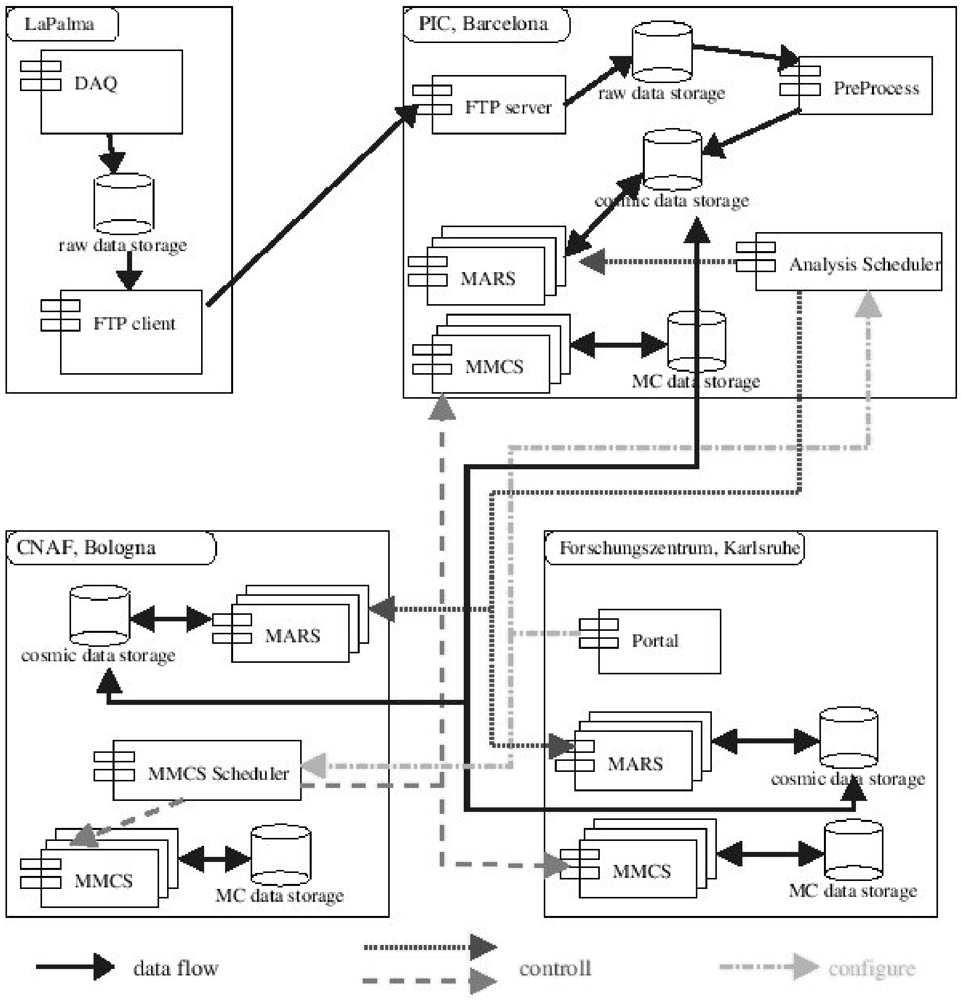


Figure 1.2: The design of the backbone for the distributed computing system shows the main components. The data from cosmic ray showers will be distributed. The corresponding dataow is drawn as the solid lines. The main two computing tasks - analysis (MARS) and monte carlo production (MMCS) - are performed at all sites. They are controlled by two schedulers located at different sites. The access to this scheduler is available via a portal from every webserver.

virtualization technologies. There are many challenges that must be addressed in order to build a working Grid environment, such as: information services, resource brokering, uniform access to resources, security, job scheduling, data access and data replication.

Grid is nowadays becoming a reality. Although it is at a very early stage of realization, many applications covering different research areas like experimental physics, astroparticle physics experiments, bioinformatics, earth observation, etc., are currently migrating to Grid.

2 Grid technologies

There is much development work to be done in order to deploy a working Grid. However, there has been a lot of work done in the past decade in the area of distributed computing and clearly it is essential to build on this wherever possible. Moreover, due to the rapid growth of the Internet and the Web, there has been a rising interest in Web-based distributed computing, and many projects have been started that aim to exploit the Web as an infrastructure for running coarse-grained distributed and parallel applications. In this context Grid is emerging as an Internet-based distributed system. An increasing number of research groups have been working in the field of wide-area distributed computing. These groups have implemented middleware, libraries and tools that allow the cooperative use of geographically distributed resources unified to act as a single powerful platform for the execution of a range of parallel and distributed applications. This approach to computing has been known by several names, such as metacomputing, scalable computing, global computing, Internet computing and lately as Grid computing. This chapter presents an overview of the technologies, open standards and protocols used for building up this new paradigm.

2.1 Brief overview of distributed systems

A general and effective definition of distributed systems can be found in [118]: *a distributed system is a collection of independent computers that appears to its users as a single coherent system.* We can mention two important characteristics: the differences between the various computers and the ways in which they communicate are hidden from users and the users and the applications can interact with a distributed system in a consistent and uniform way, regardless of where and when interaction takes place.

To support heterogeneous computers and networks while offering a single-system view, distributed systems are often organized by means of a layer of software that is logically placed between a higher-level layer consisting of users and applications, and a layer underneath consisting of operating systems. This layer of software is called **middleware**. There are many definitions of middleware. Practically, the middleware is a connectivity software that consists of a set of enabling services that allow multiple processes running on one or more machines to interact across a network. This technology has evolved during the 1990's to provide for interoperability in support of the move to client/server architectures. The most widely-publicized middleware initiatives are the Open Software Foundation's Distributed Computing Environment (DCE), Object Management Group's Common Object Request Broker Architecture (CORBA), and Microsoft's COM/DCOM (COM, DCOM).

2.1.1 Properties of a distributed system

There are many possible ways to evaluate distributed systems. Here we present a set of properties that a distributed system should implement. While not exhaustive, this set is chosen because these properties are often used when talking about the advantages or disadvantages of decentralized systems.

Security. A distributed system connects many users and resources and as this connectivity increases, security becomes more and more important. Security covers a variety of topics, such as preventing people from taking over the system, injecting bad information, or using the system for a purpose other than what the owners intend.

Transparency. A distributed system should hide the fact that its processes and resources are physically distributed across multiple computers. There are many kinds of transparency:

- *Access transparency:* hides differences in data representation and the way that resources can be accessed by users.
- *Location transparency:* manages of names for accessing resources. A physical or a logical name can be used.
- *Migration transparency:* manages the relocation of the resources.
- *Relocation transparency:* hides that a resource may be moved to another location while in use.
- *Replication transparency:* hides that a resource is replicated.
- *Concurrency transparency:* hides that a resource may be shared by several competitive users.
- *Failure transparency:* hides the failure and recovery of a resource.
- *Persistence transparency:* hides whether a software resource is in memory or on disk.

Openness. An *open distributed system* is a system that offers services according to standard rules that describe the syntax and semantics of those services. In distributed systems, services are generally specified through **interfaces**. Many problems must be addressed in order to build an open distributed system:

- Detailed interfaces of components need to be published.
- Flexibility: new components have to be integrated with existing components and it has to be easy to configure.

- Interoperability: two implementations of systems or components from different manufactures can co-exist and work together by merely relying on each others services as specified by a common standard.
- Portability: an application developed for a distributed system A can be executed, without modification, on a different distributed system B that implements the same interfaces as A.

Scalability. It indicates the capability of a system to increase total througput under an increased load when resources (typically hardware) are added. Scalability of a system can be measured along at least three different dimensions [118, 96]:

- Size: more users and resources can be easily added to the system.
- Geographic: A geographically scalable system is one that maintains its usefulness and usability, regardless of how far apart its users or resources are.
- Administrative: no matter how many different organizations need to share a single distributed system, it should still be easy to use and manage.

2.2 The client-server model

Important to any distributed system is its internal organization. The client-server model is the most widely accepted model for structuring distributed systems. A basic definition can be the following: client/server is a computational architecture that involves client processes requesting service from server processes. A **server** is a process implementing a specific service, for example, a file system service or a database service. A **client** is a process that requests a service from a server by sending it a request and subsequently waiting for the server's reply. This interaction is also known as **request-reply behavior**. In general, client/server maintains a distinction between processes and network devices. Usually a client computer and a server computer are two separate devices, each customized for their designed purpose. In any case, the same device may function as both client and server, hence, a device that is a server at one moment can reverse roles and become a client to a different server (either for the same application or for a different application).

Although communication between a client and a server can be implemented by means of a simple connectionless protocol (if the underlying network is fairly reliable), it is usually based on a reliable connection-oriented protocol, like the TCP/IP.

Advantages and disadvantages of the client-server model

The client-server model was originally developed to allow more users to share access to database applications. Compared to the mainframe approach, client-server offers improved scalability because connections can be made as needed rather than being hard-wired. Flexible user interface development is the most obvious advantage of client-server computing. It is possible to create an interface that is independent of the server hosting the data. Therefore, the user interface of a client-server application can be written on a Macintosh and the server can be written on a mainframe. Clients could be also written for DOS- or UNIX-based computers. The client-server model also supports modular applications. In the so-called *two-tier* and *three-tier* types of client-server systems, a software application is separated into modular pieces, and each piece is installed on hardware specialized for that subsystem.

One area of special concern in client-server networking is system management. With applications distributed across the network, it can be challenging to keep configuration information up-to-date and consistent among all of the devices. Therefore, upgrades to a newer version of a client-server application can be difficult to synchronize or stage appropriately. Finally, client-server systems rely heavily on the network's reliability; redundancy or fail-over features can be expensive to implement.

2.3 Communication

Interprocess communication is at the heart of all distributed systems, and there are many ways to exchange information among processes on different machines. In traditional network applications, communication is often based on the low-level message-passing primitives offered by the transport layer. An important issue in middleware systems is to offer a higher level of abstraction that will make it easier to express communication between processes than the support offered by the interface to the transport layer. At least four abstractions can be distinguished: the Remote Procedure Call (RPC), the Remote Object Invocation, the message-oriented communication and the stream-oriented communication. We briefly describe these four communication methods.

2.3.1 Remote procedure call

Remote Procedure Call (RPC) is a powerful technique for constructing distributed, client-server based applications [118, 83]. It is based on extending the notion of conventional, or local procedure calling, so that the called procedure need not exist in the same address space as the calling procedure. The two processes may be on the same system, or they may be on different systems with a network connecting them. By using RPC, programmers of distributed applications avoid the details of the interface with the network. The transport independence of RPC isolates the application from the physical and logical elements of the data communications mechanism and allows the application to use a variety of transports.

How RPC works

The idea is to make a remote procedure call look as much as possible like a local one, it has to be transparent. In order to achieve this goal, client and server **stubs** are used. When the calling process calls a procedure, the action performed by that procedure will not be the actual code as written, but code that begins network communication.

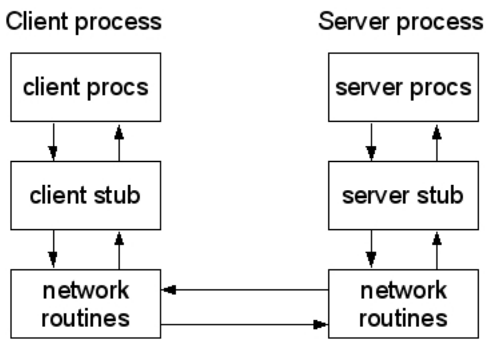


Figure 2.1: The steps involved in doing a remote computation through RPC: client and server stubs.

It has to conenct to the remote machine, send all the parameters down to it, wait for replies, do the right thing to the stack and return. This is the **client side stub**. The **server side stub** has to wait for messages asking for a procedure to run. It has to read the parameters, and present them in a suitable form to execute the procedure locally. After execution,it has to send the results back to the calling process. Stubs are usually generated automatically. The compiler must generate separate stubs, one for the client stub embedded in the application, and one for the server stub for the remote machine. In order to generate the stubs, the compiler must know which parameters are input parameters and which are output parameters. Input parameters are sent from the client to server, output parameters are sent back.

2.3.2 Remote object invocation

Distributed object-oriented systems are a natural outgrowth of object-based operating systems and languages. In these systems, every resource or abstraction is represented by an object. The key feature of an object is that it encapsulates data, called the **state**, and the operations on those data, called the **methods**. Methods are made available through an **interface**. An object may implement multiple interfaces and there may be several objects that offer an implementation for it. This strict separation between interfaces and objects allows to place an interface at one machine, while the object itself resides on another machine.

Basically, the remote object invocation is an object-oriented version of the remote procedure call [118, 83]. When a client *binds* to a distributed object, an implementation of the object's interface, called **proxy**, is loaded into the client's address space. A proxy is the equivalent of the stub in RPC. It marshals method invocations into messages and unmarshals reply messages to return the result of the method invocation to the client. The object resides at a server machine. On the server side, incoming invocation requests are first passed to a **skeleton** (the equivalent of the server side stub) which unmarshals them to proper method invocations at the object's interface at the server.

Usually it is only the interface of an object to be distributed, the state is not distributed and resides at a single machine. Such objects are also referred to as **remote objects**. It is however possible to find objects whose state may be distributed across multiple machines, and this distribution is also hidden from clients behind the object's interfaces.

2.3.3 Message-oriented communication

Message-oriented communication is a way of communicating between processes. Messages, which correspond to events, are the basic units of data delivered. Tanenbaum and Steen [118] classified message-oriented communication according to two factors: *synchronous* or *asynchronous* communication, and *transient* or *persistent* communication. In synchronous communication, the sender blocks waiting for the receiver to engage in the exchange. Asynchronous communication does not require both the sender and the receiver to execute simultaneously. So, the sender and recipient are loosely-coupled. The amount of time messages are stored determines whether the communication is transient or persistent. Transient communication stores the message only while both partners in the communication are executing. If the next router or receiver is not available, then the message is discarded. Persistent communication, on the other hand, stores the message until the recipient receives it.

There are several combinations of these types of communication that occur in practice. Examples of message oriented transient communication are the *Berkley Sockets* and the *Message Passing Interface* (MPI). On the other side, a typical example of asynchronous persistent communication is *Message-Oriented Middleware* (MOM). Message-oriented middleware is also called a *message-queuing system*, a *message framework*, or just a *messaging system.* MOM can form an important middleware layer for enterprise applications on the Internet. In the publish and subscribe model (see 2.4.1), a client can register as a publisher or a subscriber of messages. Messages are delivered only to the relevant destinations and only once, with various communication methods including one-to-many or many-to-many communication. The data source and destination can be decoupled under such a model.

2.3.4 Stream-oriented communication

Communication as discussed so far was based on:

- independent and complete units of information;
- moment of receiving is not important for correctness.

There are some situations where communication timing plays a crucial role. For example, in a real time video conference moment of receiving and correct representation are essential. To capture the exchange of time-dependent information, distributed systems generally provide support for **data streams**, which are sequence of data units. Regarding on how timing is considered, three different transmission modes can be distinguished:

- *Asynchronous transmission mode*: no timing constraints, data items are transmitted one after the other. File transmission is a typical example.
- *Synchronous transmission mode*: there is a maximum end-to-end delay for each unit in a data stream.
- *Isochronous transmission mode*: data transfer is subject to a maximum and minimum end-to-end delay, also referred to as bounded jitter. The term streams usually identifies continuous data streams using isochronous transmission.

A stream can be simple or complex, depending if it consists of only a single sequence of data or several related simple streams (e.g. stereo audio, audio and video). Substreams of complex streams must be continuously synchronized. A stream can often be considered as a virtual connection between a source and a sink. The source or sink can be a process, but could also be a device. Time-dependent (and other nonfunctional) requirements are generally expressed as *Quality of Service* (QoS) requirements. These requirements describe what is needed from the underlying distributed system and network to ensure that, for example, the temporal relationships in a stream can be preserved.

2.4 Service Oriented Architecture

This and the next section want to introduce the basic concepts on which Grid is being taking form. The first fact is that Grid is evolving into a Service Oriented Architecture (SOA), primarily based on Web Services. The SOA is based on the concept of *loose coupling.* Coupling is the dependency between interacting systems. This dependency can be decomposed into real dependency and artificial dependency:

1. Real dependency is the set of features or services that a system consumes from other systems. The real dependency always exists and cannot be reduced.
2. Artificial dependency is the set of factors that a system has to comply with in order to consume the features or services provided by other systems. Typical artificial dependency factors are language dependency, platform dependency, API dependency, etc. Artificial dependency always exists, but it or its cost can be reduced.

For example, if you travel overseas on business, you know that you must bring power adapters along with you. The real dependency is that you need power; the artificial dependency is that your plug must fit into the local outlet. Looking at all the varying sizes and shapes of those plugs from different countries, you would notice that some of them are small and compact while many others are big and bulky. We cannot remove artificial dependencies, but we can reduce them. If the artificial dependencies among systems have been reduced, ideally, to their minimum, we have achieved loose coupling.

SOA is an architectural style whose goal is to achieve loose coupling among interacting software agents. A service is a unit of work done by a service provider to

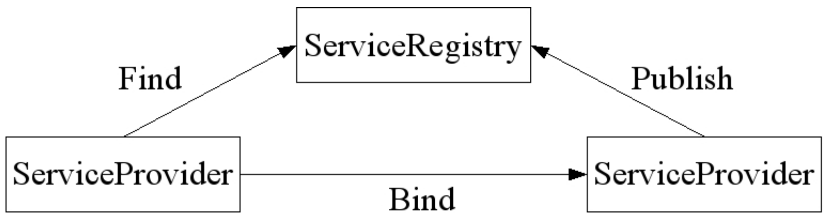


Figure 2.2: Elements of the Service Oriented Architecture

achieve desired end results for a service consumer. Both provider and consumer are roles played by software agents on behalf of their owners. How does SOA achieve loose coupling among interacting software agents? It does so by employing two architectural constraints:

1. A small set of simple and ubiquitous interfaces to all participating software agents. Only generic semantics are encoded at the interfaces. The interfaces should be universally available for all providers and consumers.

2. Descriptive messages constrained by an extensible schema delivered through the interfaces. No, or only minimal, system behavior is prescribed by messages. A schema limits the vocabulary and structure of messages. An extensible schema allows new versions of services to be introduced without breaking existing services.

Since we have only a few generic interfaces available, we must express application-specific semantics in messages. We can send any kind of message over our interfaces, but there are a few rules to follow before we can say that an architecture is service oriented. First, the messages must be descriptive, rather than instructive, because the service provider is responsible for solving the problem. Second, service providers will be unable to understand your request if your messages are not written in a format, structure, and vocabulary that is understood by all parties. Limiting the vocabulary and structure of messages is a necessity for any efficient communication. The more restricted a message is, the easier it is to understand the message, although it comes at the expense of reduced extensibility. Third, extensibility is vitally important. If messages are not extensible, consumers and providers will be locked into one particular version of a service.

2.4.1 Basic components of SOA

The SOA's basic components are the elements and the operations messages they exchange with each other. There are three key elements: Service Provider, Service Requestor and Service Registry, as shown in Figure 2.2.

Service Provider. The Service Provider is responsible for building a useful service, creating a service description for it, publishing that service description to one or more service registries, and receiving service invocation messages from one or more Service Requestors.

Service Requestor. The Service Requestor is responsible for finding a service description published to one or more Service Registries, such as yellow pages for services, and for using service descriptions to bind to or invoke services hosted by Service Providers. Any consumer of a service can be considered a Service Requestor.

Service Registry. The Service Registry is responsible for advertising service descriptions published to it by the Service Providers, and for allowing Service Requestors to search the collection of service descriptions contained within the Service Registry. Once the Service Registry provides a match between the Service Requestor and the Service Provider, the Service Registry is no longer needed for the interaction.

Operations are defined by contracts between the above elements. There are three types of contracts: Publish, Find and Bind, as shown in Figure 2.2.

Publish. The Publish operation is a contract between the Service Provider and the Service Registry. The Service Provider registers the service interfaces it provides at the Service Registry using the Publish operation. Once published, the services provided by the Service Provider are available for any Service Requestor to use.

Find. The Find operation is a contract between the Service Requestor and Service Registry. The Service Requestor uses the Find operation to get a list of the Service Providers that satisfies its needs. It may indicate one or more search criteria, such as the desired availability and performance, in the Find operation. The Service Registry searches through all the registered Service Providers and returns the appropriate information.

Bind. The Bind operation is a contract between the Service Requestor and the Service Provider. It allows the Service Requestor to connect to the Service Provider before invoking the operations. It also enables the Service Requestor to generate the client-side proxy for the service provided by the Service Provider. The binding can be dynamic or static: in the first case, the Service Requestor generates the client-side proxy based on the service description obtained from the Service Registry at the time the service is invoked; the other case involves the Service Requestor generating the client-side proxy during application development.

2.4.2 Web services as an implementation of the SOA

The W3C (World Wide Web Consortium, which develops interoperable technologies, e.g. specifications, guidelines, software, and tools, to lead the Web to its full potential) [122] gives the following definition of web service:

> A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

Several characteristics of a web service can be identified [19]:

XML based: XML is used as the data representation layer for all web services protocols and technologies.

Loosely coupled: a consumer of a web service is not tied to that web service directly; the web service interface can change over time without compromising the client's ability to interact with the service.

Coarse-grained: object-oriented technologies expose their services through individual methods. Building a program from scratch requires the creation of several fine-grained methods that are then composed into a coarse-grained service that is consumed by either a client or another service. Web services technology provides a natural way of defining coarse-grained services.

Ability to be synchronous or asynchronous: synchronicity refers to the binding of the client to the execution of the service. In synchronous invocations, the client blocks and waits for the service to complete its operation before continuing. Asynchronous operations allow a client to invoke a service and then execute other functions.

Supports Remote Procedure Calls (RPCs): web services allow clients to invoke procedures, functions, and methods on remote objects using an XML-based protocol.

Supports document exchange: One of the key advantages of XML is its generic way of representing not only data, but also complex documents. Web services support the transparent exchange of documents to facilitate business integration.

Over the past two years, three primary technologies have emerged as worldwide standards that make up the core of today's web services technology. These technologies are: SOAP, WSDL and UDDI. The Simple Object Access Protocol (SOAP) provides a standard packaging structure to transport XML documents over a variety of standard Internet technologies, including SMTP, HTTP, and FTP. It is used to exchange messages between Web services. The Web Service Description Language (WSDL) is an XML technology that describes the interface of a web service in a standardized way and allows disparate clients to automatically understand how to interact with a web service. The Universal Description, Discovery, and Integration (UDDI) provides a worldwide registry of web services for advertisement, discovery, and integration purposes. One of the big promises of web services is seamless, automatic business integration: a piece of software will discover, access, integrate, and invoke new services from unknown companies dynamically without the need for human intervention.

The web services model lends itself well to a highly distributed, service-oriented architecture (SOA). A web service may communicate with a handful of standalone processes and functions or participate in a complicated, orchestrated business process. A web service can be published, located, and invoked within the enterprise, or anywhere on the Web.

2.5 The evolution of the Grid

Three stages of Grid evolution can be identified [106]: first-generation systems born in the early to mid 1990s, second-generation systems with a focus on middleware to support large-scale data and computation and current third-generation systems in which the emphasis shifts to distributed global collaboration, a service-oriented approach and information layer issues. The first generation marked the emergence of the early metacomputing or Grid environments. Typically, the objective of these early metacomputing projects was to provide computational resources to a range of high-performance applications. Two representative projects in the vanguard of this type of technology were FAFNER [36] and I-WAY [41]. FAFNER was the forerunner of the likes of SETI@home [112] and Distributed.Net [95], and I-WAY for Globus [42] and Legion [59]. The emphasis of the early efforts in Grid computing was in part driven by the need to link a number of US national supercomputing centers.

The second-generation Grid was the result of the emergence of an infrastructure capable of binding together more than just a few specialised supercomputing centers. Now the take-up of high bandwidth network technologies and adoption of standards, allows the Grid to be viewed as a viable distributed infrastructure on a global scale that can support diverse applications requiring large-scale computation and data [43]. There are three main issues that had to be confronted: *heterogeneity*, *scalability* and *adaptability*. In a Grid, the middleware is used to hide the heterogeneous nature of the resources and to provide users and applications with a homogeneous and seamless environment by providing a set of standardised interfaces to a variety of services. Systems use varying standards and system application programming interfaces (APIs), resulting in the need to port services and applications to the plethora of computer systems used in a Grid environment. The most significant projects that have contributed to make the Grid concrete are: Globus [42], Legion [59], the European DataGrid project [121], the UNIform Interface to COmputer REsources (UNICORE) project [7], the Cactus project [6], as well as others.

With third generation there is an increasing adoption of a service-oriented model and increasing attention to metadata. There is a strong sense of automation in third-generation systems; for example, when humans can no longer deal with the scale and heterogeneity but delegate to processes to do so (e.g. through scripting), which leads to autonomy within the systems. Similarly, the increased likelihood of failure implies a need for automatic recovery: configuration and repair cannot remain manual tasks.

In the next section we describe the main characteristics of the Globus toolkit, which is the standard de facto for the Grid middleware.

2.6 The Globus Toolkit

The Globus Toolkit [42] is actually the de facto standard middleware for Grid computing. It is a *metacomputing* infrastructure toolkit providing basic capabilities and interfaces in areas such as communication, information, resource location, resource scheduling, authentication, and data access. Together, these toolkit components define a metacomputing abstract machine on which a range of alternative infrastructures can be constructed, services, and applications. The term metacomputer is used to denote a networked virtual supercomputer, constructed dynamically from geographically distributed resources linked by high-speed networks.

With version 3.0, the Globus Toolkit is a reference implementation of the Open Grid Service Architecture (OGSA) published by the Global Grid Forum (GGF). It is divided in four main components:

- The Grid Security Infrastructure.
- The resource management infrastructure (GRAM)
- The information management infrastructure.
- The data management infrastructure.

Globus Grid Security Infrastructure (GSI) Since a Grid implies crossing organizational boundaries, resources are going to be accessed by many different organizations. This poses a lot of challenges:

- We have to make sure that only certain organizations can access our resources, and that we're 100% sure that those users are really who they claim to be. In other words, we have to make sure that everyone in a Grid application is properly authenticated/authorized;
- We are going to bump into some pretty interesting scenarios. For example, suppose organization A asks B to perform a certain task. B, on the other hand, realizes that the task should be delegated to organization C. However, let's suppose C only trusts A (and not B). Should C turn down the request because it comes from B, or accept it since the 'original' requestor is A?
- Depending on the application, we may also be interested in assuring data integrity and privacy, although in a Grid application this is generally not as important as authentication.

The GSI offers the following three features:

- A complete public-key infrastructure.

- Mutual authentication through digital certificates.
- Credential delegation and single sign-on.

The GSI [129] is based on public-key cryptography, and therefore can be configured to guarantee privacy, integrity, and authentication. *Mutual* authentication is achieved using X.509 certificates [128]. Grid is a collection of heterogeneous resources that span across multiple organization domains. In this context the single sign-on is a very important feature. Situations like the one described before, where a job encompasses three different organizations, are addressed with delegation. For example, it would interesting to find a legitimate way for B to demonstrate that it is, in fact, acting on A's behalf. One way of doing this would be for A to 'lend' its public and private key pair to B. However, this is absolutely out of the question. Remember, the private key has to remain secret, and sending it to another organization (no matter how much you trust them) is a big breach in security. What can be used, instead, are certificates.

The Globus Resource Allocation Manager Globus is a layered architecture in which high-level global services are built on top of an essential set of core local services. At the bottom of this layered architecture, the Globus Resource Allocation Manager (GRAM) [27] provides the local component for resource management. Each GRAM is responsible for a set of resources operating under the same site-specific allocation policy, often implemented by a local resource management system, such as Load Sharing Facility (LSF) or Condor. GRAM provides a standard network-enabled interface to local resource management systems. Hence, computational Grid tools and applications can express resource allocation and process management requests in terms of a standard application programming interface (API), while individual sites are not constrained in their choice of resource management tools.

The *Resource Specification Language* (RSL) is used throughout this architecture as a common notation for expressing resource requirements. A variety of resource brokers implement domain-specific resource discovery and selection policies by transforming abstract RSL expressions into progressively more specific requirements until a specific set of resources is identified.The final step in the resource allocation process is to decompose the RSL into a set of separate resource allocation requests and to dispatch each request to the appropriate GRAM.

The Globus Information Management The dynamic nature of Grid environments means that toolkit components, programming tools, and applications must be able to adapt their behavior in response to changes in system structure and state [26]. Globus Metacomputing Directory Service (MDS) is designed to support this type of adaptation by providing an information-rich environment in which information about system components is always available. MDS stores and makes accessible information such as the architecture type, operating system version and amount of memory on a computer, network bandwidth and latency, available communication protocols, and the mapping between IP addresses and network technology.

An information-rich environment is more than just mechanisms for naming and disseminating information: it also requires agents that produce useful information and components that access and use that information. Within Globus, both these roles are distributed over every system component and potentially over every application. Every Globus service is responsible for producing information that users of that service may find useful, and for using information to enhance its flexibility and performance.

The Globus Data Management The main components of the data management infrastructure are:

GridFTP. It is a high-performance, secure protocol based on the Internet Engineering Task Force's FTP standards which uses the GSI (Grid Security Infrastructure) for authentication and new extensions to the FTP protocol for parallel data transfer, partial file transfer, and third-party (server-to-server) data transfer [5]. This will allow Grid applications to have ubiquitous, high-performance access to data in a way that is compatible with the most popular file transfer protocol in use today.

Data Replication. Tools for managing data replicas: multiple copies of data stored in different systems to improve access across geographically-distributed Grids and fault-tolerance [20]. These replication technologies currently include a Replica Catalog (that stores information about files and their replicas) and a Replica Management tool that combines the Replica Catalog with GridFTP to manage data replication.

2.7 Open Grid Service Architecture (OGSA)

A wide array of heterogeneous resources comprise a Grid, and it's important that they interact and behave in well-known and consistent ways. The need for open standards that define this interaction and encourage interoperability between components supplied from different sources was the motivation for the Open Grid Services Architecture (OGSA) [44], specified by the Open Grid Services Infrastructure working group of the Global Grid Forum (GGF) in June 2002. The objectives of OGSA are:

- Manage resources across distributed heterogeneous platforms.
- Deliver seamless quality of service (QoS). The topology of Grids is often complex. Interaction of Grid resources is usually dynamic. It's important that the Grid provide robust, behind-the-scenes services such as authorization, access control, and delegation.
- Provide a common base for autonomic management solutions. A Grid can contain many resources, with numerous combinations of configurations, interactions, and changing state and failure modes. Some form of intelligent self regulation and autonomic management of these resources is necessary.

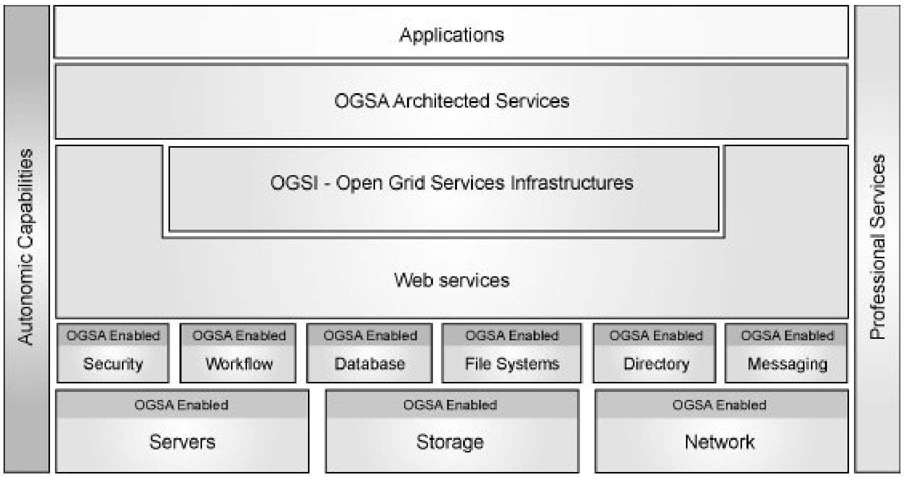


Figure 2.3: Elements of the Service Oriented Architecture

- Define open, published interfaces. OGSA is an open standard managed by the GGF standards body. For interoperability of diverse resources, Grids must be built on standard interfaces and protocols.
- Exploit industry standard integration technologies.

Figure 2.3 shows the architecture of OGSA. It comprises four main layers, starting from the bottom they are:

- Resources, physical resources and logical resources.
- Web services, plus the OGSI extensions that define Grid services.
- OGSA architected services.
- Grid applications.

Now we give a brief description of each of the four layers listed above.

Physical and logical resources Resources comprise the capabilities of the Grid, and are not limited to processors. Physical resources include servers, storage, and network. Above the physical resources are logical resources. They provide additional function by virtualizing and aggregating the resources in the physical layer. General purpose middleware such as file systems, database managers, directories, and workflow managers provide these abstract services on top of the physical Grid.

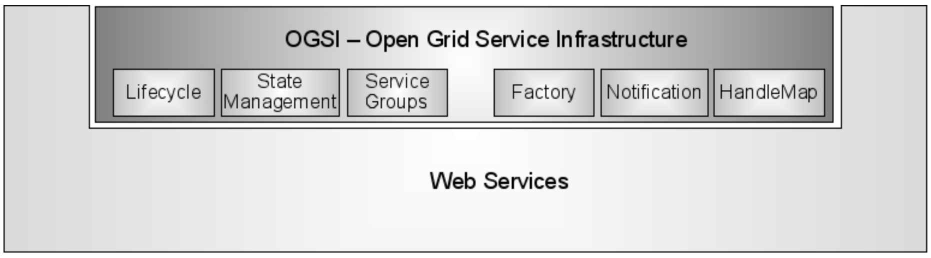


Figure 2.4: OGSA Web Services plus OGSI layer.

Web services Here's an important tenet of OGSA: all Grid resources (both logical and physical) are modeled as services. The Open Grid Services Infrastructure (OGSI) [45] specification defines Grid services and builds on top of standard Web services technology. OGSI exploits the mechanisms of Web services like XML and WSDL to specify standard interfaces, behaviors, and interaction for all Grid resources. OGSI extends the definition of Web services to provide capabilities for dynamic, stateful, and manageable Web services that are required to model the resources of the Grid.

The Web Services layer is one of the two main logical components of OGSA (the other is the OGSA architected Grid services). This layer is also the one that substantially changes in the migration to the WSRF (see section 2.8). Figure 2.4 shows a detailed view. The GGF OGSA working group believed it was necessary to augment core Web services functionality to address Grid services requirements. OGSI extends Web services by introducing interfaces and conventions in two main areas:

- First, there's the dynamic and potentially transient nature of services in a Grid. Therefore, Grid services need interfaces to manage their creation, destruction, and life cycle management.
- Second, there's state. Grid services can have attributes and data associated with them. This is similar in concept to the traditional structure of objects in object-oriented programming. Objects have behavior and data. Likewise, Web services needed to be extended to support state data associated with Grid services.

Now we briefly describe the interfaces and conventions that OGSI introduces.

Factory. Grid services that implement this interface provide a way to create new Grid services. Factories may create temporary instances of limited function, such as a scheduler creating a service to represent the execution of a particular job, or they may create longer-lived services such as a local replica of a frequently used data set. Not all Grid services are created dynamically. For example, some might be created as the result of an instance of a physical resource in the Grid such as a processor, storage, or network device.

Life cycle. Because Grid services may be transient, Grid service instances are created with a specified lifetime. The lifetime of any particular service instance can be negotiated and extended as required by components that are dependent on or manage that service. The life cycle mechanism was architected to prevent Grid services from consuming resources indefinitely without requiring a large scale distributed "garbage collection" scavenger.

State management. Grid services can have state. OGSI specifies a framework for representing this state called Service Data and a mechanism for inspecting or modifying that state named Find/SetServiceData.

Service groups. Service groups are collections of Grid services that are indexed, using Service Data, for some particular purpose. For example, they might be used to collect all the services that represent the resources in a particular cluster-node within the Grid.

Notification. The state information (Service Data) that is modeled for Grid services changes as the system runs. Many interactions between Grid services require dynamic monitoring of changing state. Notification applies a traditional publish/subscribe paradigm to this monitoring. Grid services support an interface (NotificationSource) to permit other Grid services (NotificationSink) to subscribe to changes.

HandleMap. When factories are used to create a new instance of a Grid service, the factory returns the identity of the newly instantiated service. This identity is composed of two parts, a Grid Service Handle (GSH) and a Grid Service Reference (GSR). A GSH is guaranteed to reference the Grid service indefinitely, while a GSR can change within the Grid services lifetime. The HandleMap interface provides a way to obtain a GSR given a GSH. This might seem simple, but there are a number of associated intricacies with such a request [44].

OGSA architected Grid services The previous layer, Web services with its OGSI extensions, provide a base infrastructure for this layer. The Global Grid Forum is currently working to define many of these architected Grid services in areas like program execution, data services, and core services (Figure 2.5). Some are already defined, and some implementations have already appeared. As implementations of these newly architected services begin to appear, OGSA will become a more useful service-oriented architecture (SOA).

Grid applications Over time, as a rich set of Grid-architected services continues to be developed, new Grid applications that use one or more Grid architected services will appear.

2.8 Web Services Resource Framework

In January 2004, the WS-Resource Framework (WSRF) was presented, an open framework for modeling and accessing stateful resources using Web services. WSRF defines

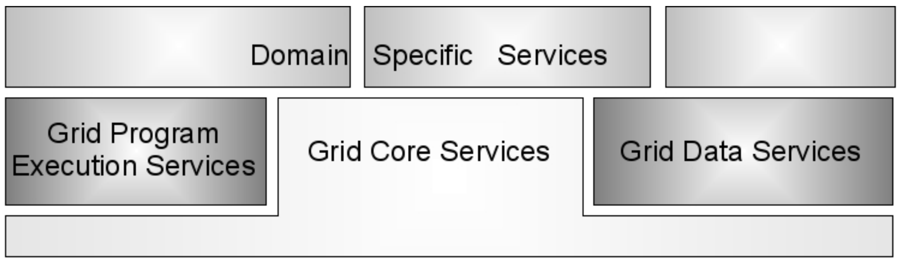


Figure 2.5: Detailed view of the OGSA architected Grid services layer.

how Web service standards are evolving to meet Grid services elements and requirements. Essentially, it represents a refactoring and evolution of OGSI (see figure 2.6) that delivers essentially the same capabilities in a manner that is more in alignment with the Web services community. The WSRF is broken up into six Web Services specifications that define terms such as the WS-Resource approach to modeling and managing state in a Web services context. The six specifications are:

- WS-ResourceProperties: specifies stateful Web services.
- WS-ResourceLifetime: specifies Web service life cycle.
- WS-RenewableReferences: specifies Web service endpoint reference and addressing.
- WS-ServiceGroup: specifies the creation and use of groups of Web services.
- WS-BaseFault: specifies fault type used for fault error reporting.
- WS-Notification: specifies the notification framework.

WSRF is the the natural convergence of the Grid services, as defined in OGSA, and the Web services framework. The main concept that arises from the migration of OGSI to WSRF is the WS-Resource, introduced to manage the state of a Grid service. If Web services are supposed to be stateless, message exchanges are, in many cases, supposed to enable access/update to state maintained by other system components; those file systems, databases or other entities, can also be considered to be stateful resources. The link between one or more stateful resources and a Web service is the Implied Resource Pattern. The Implied Resource Pattern is a set of conventions on Web services technologies, in particular XML, WSDL and WS-Addressing, implicitly meaning that the requestor does not provide the identity if the resource has an explicit parameter in the body of the request message. The context used to designate the implied stateful resource is encapsulated in the WS-Addressing endpoint reference used to address the target Web service at its endpoint. The term pattern indicates that the relationship between Web services and stateful resources is codified by a set of conventions on existing Web services technologies, XML, WSDL, and WS-Addressing.

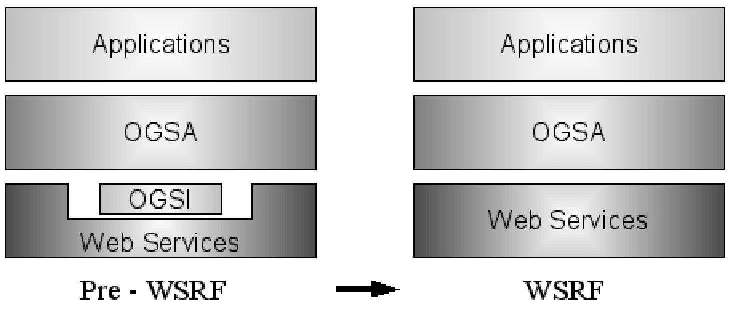


Figure 2.6: From OGSI to WSRF.

2.9 Summary

Distributed systems consist of autonomous computers that work together to give the appearance of a single coherent system. Nowadays they are generally built by means of an additional layer of software on top of a network operating system. This layer, called middleware, is designed to hide the heterogeneity and distributed nature of the underlying collection of computers. Middleware-based distributed systems generally adopt a specific model to express distribution and communication, such as: remote procedure calls, distributed objects, files, and documents.

Important to any distributed system is its internal organization. The most popular model is the client-server model. A client sends a message to a server and waits until the latter returns a reply. This model is strongly related to traditional programming, in which services are implemented as procedures or functions. Another model for structuring distributed systems is the Service Oriented Architecture (SOA). This model is receiving a growing interest. The SOA is based on the concept of *loose coupling*, where coupling is intended as the dependency between interacting systems. The SOA's basic components are elements and the operations messages they exchange with each other. There are three key elements: Service Provider, Service Requestor and Service Registry.

The Grid distributed system concept was born in the early to mid 1990s. Like any other growing idea, the Grid has evolved passing through different phases. Nowadays we are in the third Grid generation. This generation defines the Grid as a service oriented architecture based on web services. This step is characterized by the refactoring of the implementation standard OGSI that is evolved in the Web Services Resource Framework (WSRF). The main concept that arises from the migration OGSI to WSRF is the WS-Resource, introduced to manage the state of a Grid service.

3 Grid scheduling

A Grid is a dynamic heterogeneous environment that agglomerates geographically distributed resources. Therefore, a Grid scheduler (or broker) must make resource selection decisions in an environment where it has no control over the local resources, the resources are distributed, and information about the systems is often limited or dated. These interactions are also closely tied to the functionality of the Grid Information Services. In this chapter we describe properties and services characterizing the Grid Resource Management System (RMS), that provides services for the management and exploitation of the resources (scheduling services are part of the RMS). Then we tackle the problem of scheduling applications in a Grid environment and give a brief summary of current projects dealing with Grid scheduling systems.

3.1 Grid Resource Management Systems

Grid is a large, dynamic, heterogeneous and collaborative environment. Grids integrate networking, communication, computation and information to provide a virtual platform for computing and data management. Machines in a Grid are typically grouped into autonomous administrative domains that communicate via high-speed communication links. Scheduling, in such an environment, requires a different approach than presently used in distributed systems. This approach must take into account that resources in a Grid typically do not belong to the same administrative domain. Therefore, the individual demands of the participants need to be observed. This requires a new technological approach. Access to resources is typically subject to individual access, accounting, priority, and security policies of the resource owners. Moreover a *Grid resource* is a very general concept, it is not restricted only to CPU cycles, but also network bandwidth, disk space, memory, files, etc. In a broad sense a Grid resource is anything that may be used through a standard Grid interface, i.e. an Application Programming Interface and a protocol.

In a Grid environment there is the need of a set of basic functions and services that take care of all these constraints, that means accepting requests for resources and assign specific machine resources to a request from the overall pool of Grid resources for which a user has access permission. This set of services is called Grid Resource Management System (RMS). The RMS manages the pool of resources that are available to the Grid, i.e. the scheduling of processors, network bandwidth, and disk storage. In a Grid, the pool can include resources from different providers thus requiring the RMS to hold the trust of all resource providers. Basically, the RMS is constituted by the middleware, tools and services that allow to disseminate resource information, discover suitable resources and scheduling resources for job execution. The design of a RMS have to consider aspects such as: site autonomy, heterogeneity, extensibility, allocation/co-allocation, scheduling, and online control [45, 26, 10].

Site autonomy. Computing resources are geographically distributed under different ownerships each having their own access policy, cost and various constraints. Every resource owner will have a unique way of managing and scheduling resources and the Grid schedulers have to ensure that they do not conflict with resource owners policies.

Heterogeneity. The solution of complex problems can require various kind of resources located on different sites that can use different operating systems as well as different local resource management systems which lead to significant differences in functionality.

Extensibility. A resource management solution must support the frequent development of new domain-specific management structures, without requiring changes to code installed at participating sites.

Allocation/Co-allocation. Applications have computational requirements that can be satisfied by using one or more resources that could be allocated simultaneously at several sites. Site autonomy and possibility of failure during allocation introduces a need for specialized mechanisms for allocating resources, initiating computation on those resources, and monitoring and managing those computations.

Scheduling. To achieve high performance, efficient mechanism to assign job/applications tasks to the selected resources and to schedule on them their execution are needed.

Online control. Many applications can require to adapt their execution to resource availability, in particular when application requirements and resource characteristics change during application execution.

In the past several RMS were proposed, but currently no one supports a full set of functionalities as required by a Grid RMS. For example, Condor [14] supports site autonomy, but not co-allocation or online control, others, such as Legion, supports online control and policy extensibility, but not the heterogeneous substrate or co-allocation problems [30]. In [109] a taxonomy to survey existing Grid resource management implementations can be found.

Any application can be a Grid application but, in order to take full advantage from the Grid, it should be based on loosely-coupled components and it should be *Grid-aware*, i.e. designed to work in Grid. A Grid-aware application is one that at

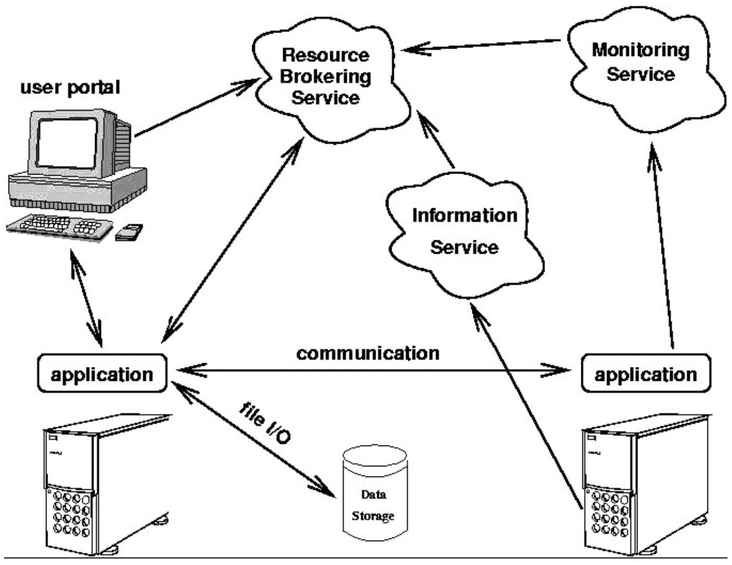


Figure 3.1: A Grid application execution scenario.

run-time can exploit the RMS to identify Grid characteristics, and then dynamically reconfigure resource requirements and possibly the application structure to maintain the required performance.

Figure 3.1 describes a Grid application execution scenario. The user submits its application to the Grid, which immediately send the application to the Resource Broker (RB). The RB is responsible to submit the user application to appropriate resources for execution. The RB uses other Grid services such as the information services and monitor services in order to select the resources and to monitor the application execution.

3.2 Introduction to Grid scheduling

Grid scheduling is defined as the process of making scheduling decisions involving resources over multiple administrative domains. This process can include searching multiple administrative domains to use a single machine or scheduling a single job to use multiple resources at a single site or multiple sites. From a Grid point of view a *job* is anything that needs a resource.

As we mentioned earlier, Grid is a collection of heterogeneous resources, in particular a computational resource can be a cluster with its own local scheduler. A Grid scheduler can exploit RMS services in order to communicate with these local schedulers. In general we can differentiate between a Grid scheduler and a *local resource scheduler*, that is, a scheduler that is responsible for scheduling and managing resources at a single site, or perhaps only for a single cluster or resource. One of the primary differences between a Grid scheduler and a local resource scheduler is that the Grid scheduler does not own the resources at a site (unlike the local resource scheduler) and therefore does not have control over them.

The structure of a scheduler may depend on the number of resources managed and the domain in which resources are located. In general we can distinguish three different models for structuring schedulers: *Centralized, Decentralized and Hierarchical* [14]. The first one can be used for managing single or multiple resources located either in a single or multiple domains. It can only support a uniform scheduling policy and suits well for cluster management (or batch queuing) systems. The Decentralized model seems to better fit for a typical Grid environment. In this model the schedulers interact among themselves in order to decide which resources should be allocated to the jobs being executed. There is no central component responsible for scheduling, hence this model appears to be highly scalable and fault-tolerant. The resource owners can define their policies that the schedulers has to enforce. However, because the status of remote jobs and resources is not available at single location, the possibility of generating highly efficient schedules is questionable. The Hierarchical model also fits for Grid environments as it allows remote resource owners to enforce their own policy on external users, and removes single centralization points. This model looks like a hybrid model (combination of central and decentralized model), and seems to better suit Grid systems. The scheduler at the top of the hierarchy is called super-scheduler/resource broker, which interacts with local schedulers in order to decide schedules.

Scheduling algorithms can be classified as *static* or *dynamic*. In the former the mapping decisions are taken before executing an application and are not changed until the end of the execution. In the latter the mapping decisions are, instead, taken while the program is running. Since static mapping usually does not imply overheads on the execution time of the mapped application, more complex mapping solutions than the dynamic ones can be adopted. The static approach is not well suited for a Grid environment, which is highly dynamic.

In order to take scheduling decisions, a Grid scheduler needs (updated) information about the available resources (e.g. architectural parameters, load, forecasts, etc.). These information are retrieved through the Grid Information System (GIS) [109], that gathers information from individual local resources. There are different information systems, designed with different architectures, but each of them deals with organizing sets of sensors in such a way that an outside system can have easy access to the data. They are designed in order to differentiate between static and dynamic data. Static data, such as type of operating system or which file systems are accessible, is often cached or made more rapidly available, whereas data that changes more often needs a heavier-weight interaction and has to be made available in very different ways (streaming versus time-out caches, for example). These system have to be extensible in order extensible to allow additional monitoring of quantities, as well as higher-level services such as better predictions or quality-of-information metrics [108]. Some examples of GIS are the Globus Monitoring and Discovery Service (MDS2) [26], the Grid Monitoring Architecture (GMA), developed by the Global Grid

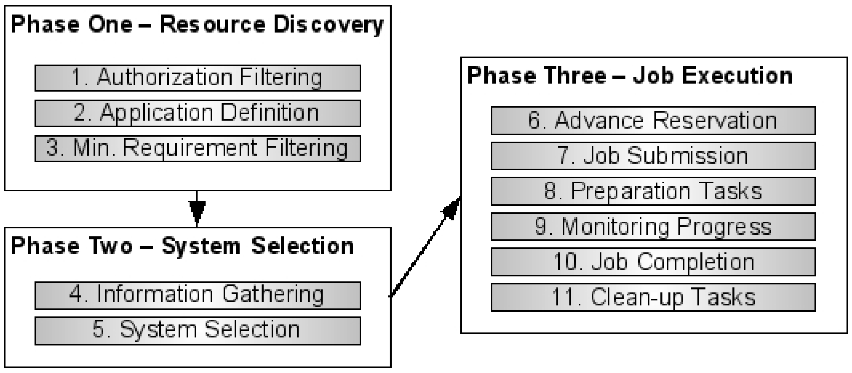

Phase One – Resource Discovery
1. Authorization Filtering
2. Application Definition
3. Min. Requirement Filtering
Phase Two – System Selection
4. Information Gathering
5. System Selection
Phase Three – Job Execution
6. Advance Reservation
7. Job Submission
8. Preparation Tasks
9. Monitoring Progress
10. Job Completion
11. Clean-up Tasks

Figure 3.2: Main phases of a Grid scheduler

Forum performance working group [123] and the Network Weather System [130]. It must be mentioned that, as a matter of fact, the decisions a scheduler makes are only as good as information provided to it.

3.2.1 General Architecture of a Grid Scheduler

In [109] Schopf delineates the general architecture of a Grid scheduler, also termed super-scheduler or broker. Grid scheduling involves three main phases (Figure 3.2): *resource discovery*, which generates a list of potential resources; *information gathering* about those resources and selection of a good set of them; and *job execution*, which includes file staging and cleanup.

Resource Discovery. This phase involves selecting a set of candidate resources and, at this moment, it is perhaps the less studied field in the scheduling community. It requires a standard way to express application requirements with respect to the resource information stored in the Grid Information System. It is thus needed an agreed-upon schema to describe the attributes of the systems, in order for different systems to understand what the values mean. An example of such a schema is the GLUE Information Model [8]. Once the list of resources is obtained, it must be filtered in order to eliminate the ones that are unusable. That means filter resources which the user submitting the job has not access to (*Authorization Filtering*), filter resources that do not match application requirements (*Application Definition*), like operating system or hardware/software requirements and finally filter resources that do not match minimal application requirements (*Min. Requirement Filtering*), with focus on dynamic requirements like load. From the above filtering criteria, the Authorization needs particular attention, it requires a secure and scalable user accounting system. G-PBox [21] and DGAS [102] are two projects addressing this problem.

System Selection. While the previous phase filters out unsuitable resources, this phase should determine from this large list the best (set of) resource(s) chosen to map the application. This selection requires to gather detailed dynamic information from resources (*Information Gathering*), e.g. by accessing local GRIS of Globus middleware, or querying performance prediction systems such as Network Weather System (NWS) [130]. This information should be used to rank resources, and to allow the scheduler to choose the ones that should ensure high performance in the execution of the application (*System Selection*).

Job Execution. The last phase is running the job. This phase can be very complex and can require various intermediary steps:

1. Advance reservation: this is an optional step. Part or all the resources needed for the job can be reserved in advance.
2. Job submission: the job is submitted to the selected resource for execution.
3. Preparation tasks: preliminary operations needed for the execution of the jobs. Can include the retrieval of input files, claiming a reservation, etc.
4. Monitoring progress: it is essential for a user to know where and how the execution is going on. These information are useful also for the scheduling algorithm, for example by rescheduling the job if it is not making sufficient progress.
5. Job completion: notification of the result of the execution.
6. Cleanup tasks: retrieving of output files, removing of temporary data and settings, and so forth.

While many schedulers have begun to address the needs of a true Grid-level scheduler, none of them currently supports the full range of actions required.

The next section gives on overview of Grid workflows, which have a growing interest and have an important impact in scheduling issues. The rest of the chapter briefly describes some implementations of scheduling systems.

3.3 Grid Workflows

Grid workflows are an emerging research field in the Grid community. Actually there is an ongoing effort to define a standard meaning of workflow for the Grid. This section wants to outline motivations, issues and approaches currently followed.

We said above how Grid is emerging as a service oriented architecture. In particular, the “software as a service” approach results in a componentized view of software applications and workflow can naturally be used as a component composition mechanism. Traditionally, the main applications of workflow have been in the automation of administrative and production processes, especially within businesses and large organizations. In the late 1990s, Enterprise Application Integration (EAI) emerged as a new application area for workflow. The expansion of workflow towards middleware has since continued, namely within the Web service initiative. The Grid can directly leverage workflow, both in terms of models and technology.

The users have complex tasks and want to take advantage of the resource-rich environment provided by the Grid to solve their problems subject to a set of constrains such as deadlines, cost, quality of the solution. A complex task consists of multiple activities. *Activities* are units of work to be performed by the agents, humans, computers, sensors, and other man-made devices involved in the workflow enactment. A *process description*, also called a *workflow schema*, is a structure describing the activities to be executed and the order of their execution [91]. A workflow has three dimensions:

1. *The process*: the process dimension refers to the creation and the possible modification of the process description.

2. *The case*: the case dimension refers to a particular instance of the workflow when the attributes required by the process enactment are bound to specific values.

3. *The resource*: the resource dimension refers to discovery and allocation of resources needed for the enactment of a case.

Workflow enactment is the process of carrying out the activities prescribed by the process description for a particular case.

Two types of workflows can be distinguished: *static* and *dynamic*. The process description of a static workflow is invariant in time. The process description of a dynamic workflow changes during the workflow enactment phase due to circumstances unforeseen at the process definition time.

There are several distinctions between Grid-based workflows and traditional workflows encountered in business management, office automation, or production management [92]:

- The emphasis in a traditional workflow model is placed on the contractual aspect of a transaction. For a Grid-based workflow the enactment of a case is sometimes based on a "best-effort model" where the agents involved do their best to attain the goal state but there is no guarantee of success.

- An important aspect of a transactional model is to maintain a consistent state of the system. A Grid is an open system, thus, the state of a Grid is considerably more difficult to define than the state of a traditional system.

- A traditional workflow consists of a set of well-defined activities that are unlikely to be altered during the enactment of the workflow. However, the process description for a Grid-based workflow may change during the lifetime of a case. After a change, the enactment of a case may continue based on the older process description, while under some circumstances it may be based on the newer process description. In addition to static workflows we have to support dynamic ones.

- The activities of a Grid-based workflow could be long lasting. Some of the activities supported by the Grid are collaborative in nature and the workflow management should support some form of merging of partial process descriptions.
- The individual activities of a Grid workflow may not exhibit the traditional properties of transactions. Consider, for example, durability; at any instance of time before reaching the goal state a workflow may roll back to some previously encountered state and continue from there on an entirely different path. An activity of a Grid workflow could be either reversible or irreversible. Sometimes, paying a penalty for reversing an activity is more profitable in the long run than continuing on a wrong path.
- Resource allocation is a critical and very delicate aspect of the Grid-based workflow enactment. The Grid provides a resource-rich environment with multiple classes of resources and many administrative domains; there is a large variability of resources in each class; resource utilization is bursty in nature. Thus, we need: resource discovery services, support for negotiations among multiple administrative domains, matching and brokerage services, reservations mechanisms, support for dynamic resource allocation, and other sophisticated resource management mechanisms and services.
- Mobility of various agents involved in the enactment of a case is important for Grid-based workflows. The agents may relocate to the proximity of the sites where activities are carried-out to reduce communication costs and latency.

At least four important issues can be identified in order to enable workflows for the Grid:

1. User Environments or Workflow IDE (Integrated Development Environment).
2. Representation and language to express workflow.
3. Translation or compilation.
4. Execution and runtime support.

Actually, the most common Grid workflow can be modelled as simple Task (Directed Acyclic) Graphs (DAGs), where the order of execution of tasks (modelled as nodes) is determined by dependencies (in turn modelled as directed arcs). Each DAG node represents the execution of a component, characterized by a set of attributes such as an estimate of its cost and possible requirements on the target execution platform, while DAG directed edges represent data dependencies between specific application components. Data dependencies will be usually constituted by large files written by a component and required for the execution of one or more other components of the application.

In the following section we will survey the main approaches to scheduling systems, that considers workflow scheduling, currently followed in the Grid community.

3.4 Grid scheduling systems

In this section we present a brief description of some Grid scheduling systems that manages workflows [136, 10].

3.4.1 Condor DAGMan

Condor [86, 119, 120] is a specialized resource management system (RMS) developed at the University of Wisconsin-Madison for compute-intensive jobs. Like other full-featured batch systems, Condor provides a job queuing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Users submit their serial or parallel jobs to Condor, Condor places them into a queue, chooses when and where to run the jobs based upon a policy, carefully monitors their progress, and ultimately informs the user upon completion. Condor can be used to manage a cluster of dedicated compute nodes (such as a "Beowulf" cluster). In addition, unique mechanisms enable Condor to effectively harness wasted CPU power from otherwise idle desktop workstations. Some of the enabling mechanisms of Condor are:

- *ClassAds*: it is a framework for matching resource requests (e.g. jobs) with resource offers (e.g. machines). It allows jobs and resources to advertise their characteristics and requirements.

- *Job checkpoint and migration*: with particular types of jobs, Condor can transparently save the state and subsequently resume the application from the checkpoint file. This technique also permits a job to migrate from one machine to another machine (low-penalty preemptive-resume scheduling [72]).

- *Remote system calls*: it is a mechanism for preserving the local execution environment. By redirecting the I/O, for example, the user do not need to make data files available on remote machines, even in the absence of a shared file system.

Condor-G [48] represents the link between two technologies: Condor and the Globus toolkit [42] (see 2.6). With Condor-G it is possible to use Condor inside a Grid environment. Condor-G can be used as a reliable submission and job management service for one or more sites, Condor as the fabric management service and finally the Globus toolkit can be used as the bridge between them.

Another service of Condor is the Directed Acyclic Graph Manager (DAGMan) [120] for executing multiple jobs with dependencies described as DAGs. Each job is a node in the graph and the edges identify their dependencies. DAGMan does not support automatic intermediate data movement, so users have to specify data movement transfer through pre-processing and post-processing commands associated with processing job. The DAGMan meta-scheduler processes the DAG dynamically, by sending to the condor scheduler the jobs as soon as their dependencies are satisfied and they become ready to execute.

3.4.2 GrADS

The Grid Application Development Software (GrADS) project [11] aims to provide programming tools and execution environments for ordinary scientific users to develop, execute, and tune applications on the Grid. To achieve this goal they are conducting research in four main areas:

1. collaboration on the design and implementation in prototype form of important scientific applications for the Grid;

2. design of programming systems and problem-solving environments;

3. design and implementation of schedulers that dynamically match configurable applications to available resources;

4. design and construction of hardware and software testbeds for experimentation.

In GrADS applications are encapsulated as *configurable object programs* (COPs). A COP includes *code* for the application (e.g. an MPI program), a *mapper* that determines how to map an application's tasks to a set of resources, and an executable *performance model* that estimates the application's performance on a set of resources. The system relies upon *performance contracts* that specify the expected performance of modules as a function of available resources.

The GrADS project has recently included the possibility of scheduling workflow applications [25]. The implemented workflow scheduling has the objective of minimize the overall job completion time, or *makespan*, of the application. The mapping from the workflow components to the Grid resources is based on the merging of two models: the model of the Grid resources, determined using Grid services such as the MDS [39] and NWS [130], and the performance model of the application. For each application component, the GrADS workflow scheduler ranks each eligible resource, reflecting the fit between the component and the resource. After ranking the components, the scheduler uses these information to build a performance matrix and, then, it runs heuristics on this matrix in order to optimize the mapping of independent jobs when several DAG nodes become runnable upon the scheduling of their parents in the graph. Three heuristics have been applied in GrADS; those are Min-Min, Max-Min, and Sufferage heuristics [89].

The estimate of the performance of a workflow component on a machine is the base for a good workflow schedule. Performance models are obtained at compile time, when the characteristics of the resources on which the components will run are not known. Therefore, performance models do not aim to predict an exact execution time, but rather provide an estimate resource usage that can be converted to a rough time estimate based on architectural parameters. To obtain the component models, they consider both the number of floating point operations executed and the memory access pattern.

3.4.3 UNICORE

UNICORE plus [125] provides seamless and secure access to distributed resources of the German high performance computing centers. UNICORE plus is a follow-on project of UNICORE (Uniform Interface to Computing Resources) [7] that offers a ready-to-run Grid system including client and server software. The original UNICORE job model supports jobs that are constructed as a set of directed acyclic graphs with temporal dependencies. Subsequent versions introduced more sophisticated control facilities in the workflow language. They have introduced advanced flow controls such as Do-N, Do-Repeat, If-then-else, and Hold-Job. UNICORE plus offers a graphical user interface to construct the application workflow and to convert it into an executable object. UNICORE also allows users to explicitly specify the transfer function as a task through GUI; it is also able to perform the necessary data movement function without user intervention.

3.5 Summary

The growing computational power requirements of grand challenge applications has promoted the need for linking high-performance computational resources distributed across multiple organizations. This is fueled by the availability of the Internet, the growing performance of the network infrastructure, low cost high-performance machines such as clusters across multiple organizations, and the rise of scientific problems of multi-organizational interest.

Computational Grids are expected to offer dependable, consistent, pervasive, and inexpensive access to high-end resources irrespective of their physical location and the location of access points. Due to the use of geographically distributed multiorganizational resources, the Grid computing environment needs to dynamically address issues involved in inter-domain resource usage and should have the following features: site autonomy, heterogeneity, extensibility, allocation/co-allocation, scheduling, and online control.

A Grid scheduler (or broker) must make resource selection decisions in an environment where it has no control over the local resources, the resources are distributed, and information about the systems is often limited or dated. These interactions are also closely tied to the functionality of the Grid Information Services. This Grid scheduling approach has three phases: resource discovery, system selection, and job execution.

Scientists and engineers are building more and more complex applications to manage and process large data sets, and execute scientific experiments on distributed resources. Such application scenarios require means for composing and executing complex workflows. Grid is emerging as a service oriented architecture. In particular, the "software as a service" approach results in a componentized view of software applications and workflow can naturally be used as a component composition mechanism. Two types of workflows can be distinguished: *static* and *dynamic*. The process description of a static workflow is invariant in time. The process description of a dynamic workflow changes during the workflow enactment phase due to circumstances unforeseen at the process definition time. Actually, the most common Grid workflow can be modelled as simple Task (Directed Acyclic) Graphs (DAGs), where the order of execution of tasks (modelled as nodes) is determined by dependencies (in turn modelled as directed arcs).

In the recent past, several Grid workflow systems have been proposed and developed. Some examples are the Condor DAGman, the GrADS project, the UNICORE project.

4 Literature survey of the DAG scheduling problem

In this chapter we present background and a brief survey of the most representative DAG scheduling algorithms found in the literature. We first introduce the DAG scheduling problem giving a general description of the problem statement. This is followed by a discussion about the NP-completeness of variants of the scheduling problem. We describe the DAG model of a parallel program and the computing system environment which executes the application. A lot of (earlier) works make simplifying assumption like restricted graph structures, no communication among tasks, etc. We discuss the difficulties in considering the general problem, in particular in presence of communication: the concept of granularity of a DAG is defined. In order to introduce the description of the most representative scheduling algorithms, we illustrate a taxonomy based on the classification of different scheduling environments. Most of the algorithms work on homogeneous and heterogeneous systems. However, the heterogeneity is mostly intended as machine heterogeneity and do not consider network heterogeneity. This means that a message is transmitted with the same speed on all links. This assumption cannot be made in a Grid environment. Some recent works on scheduling parallel applications in heterogeneous systems with a heterogeneous interconnection network are presented.

4.1 The DAG scheduling problem

The scheduling problem is very important for the effective utilization of distributed computer systems. The general scheduling problem can be divided into two categories: independent jobs scheduling and multiple interacting jobs scheduling. In the former category, independent jobs are to be scheduled among the processors of a distributed computing system to optimize the overall system performance. The latter category requires the allocation of multiple interacting tasks (considered like a single parallel program) without violating precedence constraints in order to minimize the completion time on the parallel computer system. The scheduling problem can be addressed in both static as well as dynamic contexts. When the characteristics

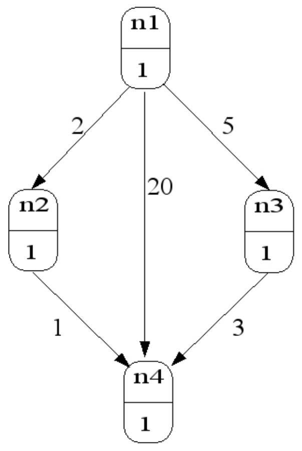


Figure 4.1: Example of precedence-constrained task graph.

of the parallel program (including its task execution times, task dependencies, task communications and synchronization) and the characteristics of the system (machines and networks specifications like cpu, memory, latency and bandwidth) are known a priori, scheduling can be accomplished offline during compile-time. On the contrary, dynamic scheduling is required when a priori information is not available and scheduling is done on-the-fly according to the state of the system.

In this thesis we consider the problem of scheduling parallel applications formed by multiple interacting jobs in a dynamic environment formed by a set of heterogeneous resources and a non-uniform interconnection network. In this context a parallel application is generally modeled by a precedence-constrained task graph, which is a directed acyclic graph (DAG) with node and edge weights (the nodes represent the tasks and the directed edges represent the execution dependencies as well as the amount of communication). In this model a task cannot start execution before all of its parents have finished their execution and sent all of the messages to the machine assigned to that task. Figure 4.1 shows an example of precedence-constrained task graph where task n_4 cannot start before tasks n_1, n_2 and n_3, edge labels are data transfer times and node labels are computation times. The scheduling objective is to minimize the program completion time.

Scheduling of precedence-constrained task graphs is an NP-complete problem in its general form [51, 68, 80]. To tackle the problem, simplifying assumptions have been made regarding the task graph structure representing the program and the model for the parallel processor system. However, the problem is NP-complete even in two simple cases: (1) scheduling unit-time tasks to an arbitrary number of processors [58], (2) scheduling one or two time unit tasks to two processors [22]. There are only three special cases for which optimal polynomial-time algorithms exist. These cases are: scheduling tree-structured task graphs with identical computation costs on an arbitrary number of processors, scheduling arbitrary task graphs with identical computation costs on two processors [60, 111] and, finally, scheduling an *interval-ordered* DAG with uniform node-weights to an arbitrary number of processors [99]. A DAG is called interval-ordered if every two precedence-related nodes can be mapped to two non-overlapping intervals on the real number line [38]. However, even in these cases, no communication is assumed among the tasks of the parallel program. It has been shown that scheduling an arbitrary task graph with inter-task communication onto two processors is NP-complete and scheduling a tree-structured task graph with inter-task communication onto a system with an arbitrary number of processors is also NP-complete [84].

In order to solve the scheduling problem, three approaches have been suggested: state-space search, dynamic programming and heuristics. The first two techniques give optimal solutions under relaxed constraints, but they are not useful since most of them work under restricted environments and most importantly they lead to an exponential time in the worst case. Heuristics are the usual approach to solve the scheduling problem. They achieve good efficiency, but, on the other side, they usually cannot generate optimal solutions and there is no guarantee about their performance in general, since the average, worst, and best case performance of these algorithms are not known.

Most scheduling heuristics algorithms are based on the *list scheduling* approach. The basic idea in list scheduling is to assign priorities to nodes and examine for scheduling higher priority nodes first, ties are broken using some method specific to each algorithm. There are numerous variations in how the priority are assigned and in maintaining the list of ready nodes, and criteria for selecting a processor to accommodate a task. We describe later in this chapter (section 4.6) some of these algorithms. There are two approaches used in the list scheduling: static and dynamic. In static list scheduling the list of nodes is statically constructed before task allocation begins, and the sequencing in the list is not modified during the other operations. In dynamic list scheduling priorities are updated after each allocation and the list of ready tasks is consequently rearranged. Dynamic list scheduling algorithms can potentially generate better schedules (at the cost of increasing the time-complexity), in particular if they have to operate in a Grid environment, which is extremely dynamic. In a Grid the resources are shared by all the users and the load of the network and of the machines change also during application execution. Moreover, the resources can come and go and the initial static schedule can become completely useless.

In realistic cases, a scheduling algorithm needs to address a number of issues. It should exploit the parallelism by identifying the task graph structure, and take into consideration task granularity (amount of computation with respect to communication), arbitrary computation and communication costs. Moreover, in order to be of practical use, a scheduling algorithm should have low complexity and should be economical in terms of the number of processors used.

Authors	Complexity	Processors	Computation	Graph	Comm.
Hu [60]	$O(v)$	Unlimited	Uniform	Free-tree	No
Coffman and Graham [23]	$O(v^2)$	2	Uniform	Any	No
Sethi [111]	$O(v\alpha(v) + e)$	2	Uniform	Any	No
Papadimitriou and Yannakakis [99]	$O(v + e)$	Unlimited	Uniform	Interval-Ordered	No
Ali and El-Rewini [4]	$O(ve)$	Unlimited	Uniform	Interval-Ordered	Uniform
Papadimitriou and Yannakakis [99]	NP-Complete	Unlimited	Any	Interval-Ordered	No
Garey and Johnson [51]	Open	Fixed, > 2	Uniform	Any	No
Ullman [124]	NP-Complete	Unlimited	Uniform	Any	No
Ullman [124]	NP-Complete	Fixed, > 1	$= 1$ or 2	Any	No

Table 4.1: Optimal solutions for the DAG scheduling problem under simplified situations: the table considers the time complexity of the algorithm, the number of processors of the environment, computation characteristics, graph structure and communication characteristics.

4.1.1 NP-completeness of the DAG scheduling problem

The DAG scheduling problem is NP-complete and polynomial time solutions were found only for simple cases. As we mentioned above there are only three simple cases for which the DAG scheduling problem have been solved in polynomial time. Table 4.1 summarizes these optimal results.

The first case is to schedule tree-structured task graphs with identical computation costs to an arbitrary number of processors. Hu [60] proposed a linear time solution for this problem. The second case is to schedule arbitrary task graphs with identical computation costs to two processors. Coffman and Graham [23] presented a quadratic-time algorithm to solve this problem. Sethi [111] improved the algorithm proposed by Coffman and Graham with an almost linear-time solution. The third case is to schedule an *interval-ordered* DAG with uniform node weights to an arbitrary number of processors. To solve this problem Papadimitriou and Yannakakis found a linear-time algorithm [99]. Ali and El-Rewini [4] showed a polynomial-time solution to schedule interval-ordered DAGs with uniform edge weights, which are equal to the node weights.

In its general formulation the DAG scheduling problem is NP-complete [51, 51, 68, 80, 76]. Ullman showed that scheduling a DAG with unit computation costs to p processors, and scheduling a DAG with one or two unit computation costs to two processors are NP-complete [22, 124]. Papadimitriou and Yannakakis showed that scheduling an interval ordered DAG with arbitrary computation costs to two processors is NP-complete [99] and scheduling a DAG with unit computation costs to p processors, possibly with task-duplication, is also NP-complete [100]. Finally, Garey *et al.* [50] showed that scheduling an opposing forest with unit computation to p processors is NP-complete.

4.2 Background

A parallel program can be represented by a directed acyclic graph (DAG). A DAG is a tuple $G = (V, E)$ where $V = \{n_j, j = 1..v\}$ is the set of nodes with $|V| = v$, E is the set of communication edges and $|E| = e$. Nodes and edges are labeled as follows:

- the weight $c_{i,j} \in C$ is the communication cost incurred along the edge $e_{i,j} = (n_i, n_j) \in E$, which becomes zero if both nodes are mapped to the same processor;

- the weight $\tau_i \in T$ of the node $n_i \in V$ is the computation cost (or the expected execution time) of the task represented by the node n_i.

A task is a set of instructions that must be executed sequentially in the same processor without preemption. A task cannot start execution before all of its parents have finished their execution and sent all of the messages to the machine assigned to that task. Hereafter we use the terms node and task interchangeably. Two tasks are called **independent** if there are no dependence paths between them. The **width** of a DAG is the size of the maximal set of independent tasks. A node is a **source node** if it has no incoming edges and it is an **exit node** if it has no outgoing edges.

DAG scheduling algorithms found in the literature usually assume as starting point a DAG labeled with estimates of computation times and data transfer times. In practice we have to distinguish between two types of DAGs: abstract and concrete. The **abstract DAG** is labelled with relative values, which do not aim to predict an exact execution time, but rather provide an estimated resource usage that can be converted to a rough time estimate based on architectural parameters. The abstract DAG is obtained at compile-time. The **concrete DAG** is labelled with time estimates for the execution of the tasks and the transfer of the data. An initial static matching and scheduling phase is necessary in order to obtain the concrete DAG. As noted in [90], the initial mapping can be realized using a static mapping algorithm such as the baseline [126], genetic-algorithm-based mapper [126] or Levelized Min Time [64]. In particular, when we are dealing with a heterogeneous and dynamic environment, time estimates are not known prior to application execution and the initial mapping must be done at run-time before the execution of the scheduling algorithm. All the algorithms described in the following sections assume the initial DAG to be concrete.

If we denote with $ST(n_i)$ and with $FT(n_i)$, respectively, the start and finish-time of the task n_i, then the **schedule length** can be defined as $\max_i\{FT(n_i)\}$. The objective of DAG scheduling is to minimize of the schedule length or **makespan**

$\max_i\{FT(n_i)\}$, without violating precedence constraints. Just as qualitative definition, a schedule is considered *efficient* if the scheduled length is short and the number of processors used is reasonable.

Node and edge weights are usually obtained by estimation once the characteristics of the environment (cpu, memory, latency, bandwidth, etc.) are known. Estimates are computed by considering information such as number of floating point operations and memory access pattern. Obtaining these estimates is actually an active area of research. For example, considering Grid context, we can find the GrADS project [25], where performance models are built at compile time for each task. Performance models are an architecture-independent model of the workflow component providing an estimated resource usage that can be converted to a rough time estimate when architectural parameters of the resource will be available.

The DAG scheduling problem is usually identified by the combination of two phases: **matching** which assigns tasks to machines and **scheduling** which defines the execution order of the tasks assigned to each machine. The overall problem of matching and scheduling is referred to as **mapping**.

4.2.1 The computing system

The computing system, which executes the parallel application, is generally assumed to be a network of processing elements (PEs), each of which is composed of a processor and a local memory unit so that the PEs do not share memory and communication relies solely on message-passing. There can be two types of computing system: homogeneous and heterogeneous. A heterogeneous computing system consists of a heterogeneous suite of machines having different speeds or processing capabilities. The machines are connected by a network. The topology of the network may be fully-connected or of a particular structure such as a hypercube or a mesh. Although most of the scheduling algorithms that can be found in the literature assume an environment with heterogeneous processors, usually the communication links are assumed to be homogeneous, i.e., a message is transmitted with the same speed on all links. In contrast, a Grid has both heterogeneous machines and communication links. The data communication time between two machines has two components: a fixed message latency for the first byte to arrive and a per byte message transfer time. In such situations, an $|M| \times |M|$ communication matrix, where $|M|$ is the number of machines, is used to hold these values for the heterogeneous computing (HC) suite.

4.2.2 DAG scheduling preliminaries

As we mentioned above, most of the scheduling algorithms are based on the **list scheduling** technique. These algorithms assign priorities to the tasks and schedule them according to a list priority scheme. A node with higher priority is examined for scheduling before a node with lower priority. This technique is based on the repeated execution of the following two steps for as long as all the tasks of the DAG are mapped:

1. select the node with higher priority;

```
Create TList, a list of nodes in topological order.
foreach node n of TList do
  max = 0
  foreach parent p of n do
    if (tlevel(p) + $\tau_p$ + $c_{p,n}$) > max then
      max = tlevel(p) + $\tau_p$ + $c_{p,n}$
    endif
  endfor
  tlevel(n) = max
endfor
```

Listing 4.1: Procedure to compute the tlevel.

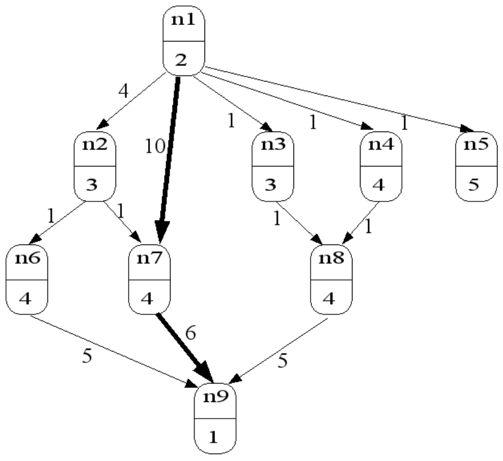
```
Create RTList, a list of nodes in reversed topological order.
foreach node n of RTList do
  max = 0
  foreach child c of n do
    if ($c_{n,c}$ + blevel(c)) > max then
      max = $c_{n,c}$ + blevel(c)
    endif
  endfor
  blevel(n) = $\tau_n$ + max
endfor
```

Listing 4.2: Procedure to compute the blevel.

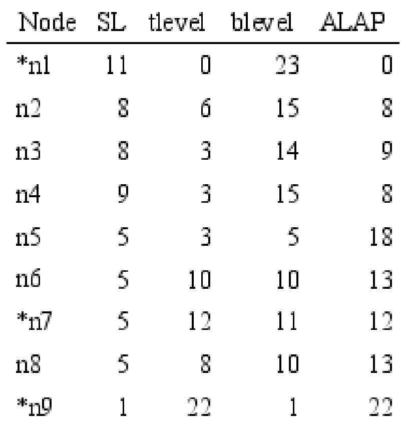
2. assign the selected node to a suitable machine.

The various list scheduling algorithms differ in the methods of assigning priorities and maintaining the ready list, and criteria for selecting a processor to accommodate a node. Traditionally the list of nodes is statically constructed before node allocation begins. In *dynamic* list scheduling priorities are re-computed before the selection phase. This step is very important for dynamic environments and a lot of care must be taken since it can increase the time-complexity of the scheduling algorithm. We describe below some methods used for assigning priorities.

Two major attributes frequently used for assigning priorities are the *tlevel* (top level) and the *blevel* (bottom level). The **tlevel** of a node n_i is the weight of the longest path from a source node to n_i (excluding n_i). The length of a path is computed as the sum of all the node and edge weights along the path. In the computation of the *tlevel* of a node n_i its execution time τ_i is not included. The *tlevel* of n_i identifies the n_i's earliest start time, denoted by $T_S(n_i)$, which is determined after n_i is mapped to a machine. It is a dynamic attribute because the weight of an edge may be zeroed when the two incident nodes are mapped to the same processor. Some authors call this attribute **ASAP** (As Soon As Possible). Listing 4.1 shows a procedure to compute



(a)



Node	SL	tlevel	blevel	ALAP
*n1	11	0	23	0
n2	8	6	15	8
n3	8	3	14	9
n4	9	3	15	8
n5	5	3	5	18
n6	5	10	10	13
*n7	5	12	11	12
n8	5	8	10	13
*n9	1	22	1	22

(b)

Figure 4.2: (a) A DAG. (b) The static level SL, tlevel, blevel and ALAP of the nodes.

the *tlevel* in $O(v + e)$ time-complexity.

The **blevel** (bottom level) of a node n_i is the weight of the longest path from n_i (this time τ_i is included) to an exit node. Listing 4.2 shows a procedure to compute the *blevel* in $O(v + e)$ time-complexity.

The **critical path** (CP) of a DAG is the longest path in that graph, i.e. the path whose length is the maximum. There can be more than one CP. The *tlevel* and the *blevel* are bounded from above by the length of the critical path.

Most of the scheduling algorithms do not allow the mapping of a child before its parents. In this case the *blevel* of a task is a static attribute and does not change until after the task is mapped to a machine. The procedure in listing 4.2 computes the *blevel* using the edge weights in the computation of the heaviest path from the considered node to an exit node. Some algorithms do not take into account the edge weights. In this case it is called the *static blevel* or simply static level (SL). In general, scheduling in descending order of *blevel* tends to schedule critical path nodes first, while scheduling in ascending order of *tlevel* tends to schedule nodes in topological order.

A parameter related to the ASAP is the **ALAP** (As Late As Possible), the ALAP of a node n_i is defined as the difference between the weight of the CP of the DAG and the *blevel* of n_i. The ALAP measures how far the node's start-time can be delayed without increasing the schedule length. Some algorithms use the value given by the difference between ALAP and ASAP in order to assign priorities to the tasks. This value is called the *mobility* of a node. It is sometimes used to schedule the tasks assigned to the same processor. The execution of already scheduled tasks can be delayed in order to create a time-slot for accommodate a new task (given that precedence-constraints are not violated). The amount of delay for a specific task is

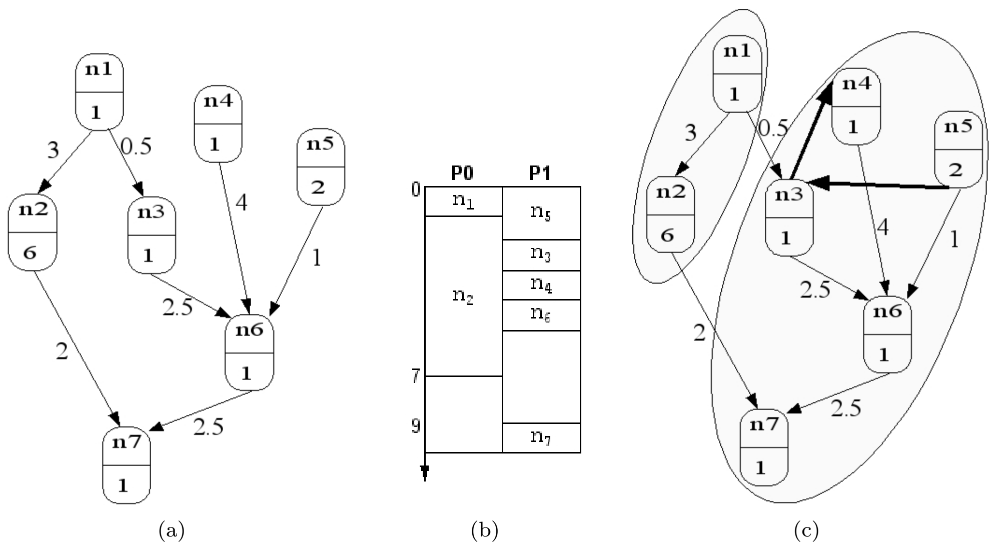


Figure 4.3: (a) A DAG. (b) A Gantt chart for a schedule. (c) A scheduled DAG

given by its mobility attribute.

In figure 4.2 we can see an example of DAG where nodes n_1, n_7 and n_9 belong to the critical path. In this case the critical path is unique. The edges of the critical path are shown with thick arrows. If we change, for example, the weight of the edge (n_8, n_9) from 5 to 10, then also the nodes n_1, n_4, n_8 and n_9 lie on a critical path, two CPs of weight 23.

Figure 4.3(a) shows an example of weighted DAG, figure 4.3(b) a Gantt chart for a possible schedule of that DAG and figure 4.3(c) the corresponding scheduled DAG. The **Gantt chart** defines the processor assignment, the starting and the completion times for each task. A **scheduled DAG** is a DAG with included precedence constraints given an assigned schedule. In figure 4.3(c) the two clusters are circled and the edges (drawn with thick lines) corresponding to task execution ordering information for tasks assigned to the same processor are: (n_3, n_4) and (n_5, n_3).

A cluster with only one task is called a **unit cluster**. Clusters can be divided into two groups: *linear* and *nonlinear*. A cluster is called **nonlinear** if there are at least two independent tasks in the same cluster, otherwise it is called **linear**. For example, in figure 4.3(c) there are two clusters: one is a linear cluster and contains the nodes n_1 and n_2, the other is nonlinear because nodes n_3, n_4 and n_5 are independent in the original DAG (figure 4.3(a)). Since a schedule imposes an ordering of tasks in nonlinear clusters, the nonlinear clusters of a DAG can be thought of as linear clusters in the scheduled DAG if execution orders between independent tasks are counted as edges. It is important to note that liner clusters preserve the parallelism of the DAG while nonlinear clusters reduces parallelism by sequentializing parallel tasks. The core point of the DAG scheduling problem is to find the best tradeoff

Symbol	Definition
V	The set of nodes (tasks) in the DAG.
E	The set of edges (communication links) in the DAG.
n_i	Node (task) of the DAG ($n_i \in V$).
$e_{i,j}$ or (n_i, n_j)	Edge from task n_i to task n_j ($e_{i,j \in E}$).
τ_i	The execution time of node (task) n_i.
$c_{i,j}$	The communication time between the tasks.
CCR	Communication to computation ratio.
$tlevel(n_i)$	Top level of task n_i (earliest start time).
$blevel(n_i)$	Bottom level of task n_i.
CP	Critical Path (longest path in the DAG).
DS	Dominant Sequence (critical path of the scheduled DAG).
$ASAP(n_i)$	As Soon As Possible (same as $tlevel(n_i)$ or earliest start time).
$ALAP(n_i)$	As Late As Possible.

Table 4.2: Some notations and their definitions

between sequentialization and parallelization. A discussion about this issue can be found in [54]. In section 4.2.3 we briefly summarize that work.

A task is **free** if it has no predecessors or if all its predecessors have been examined and mapped to a resource. A task is **partially free** if some of its predecessors have been examined and mapped to a resource.

A **ready task** is a task that can immediately start its execution because all the data it is waiting for (from its parents) are available on the assigned machine. A task is **not ready** if it is still waiting for data. A ready task is also a free task, but a free task can be ready or not.

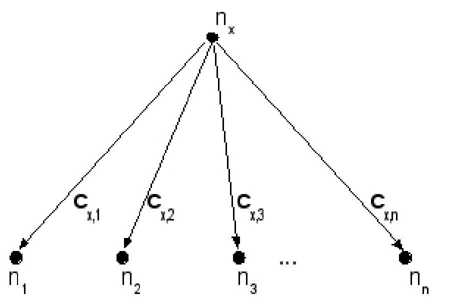
The **Dominant Sequence** (DS) is the critical path of the scheduled DAG and its weight is called the **parallel time**. The following formula can be used to determine the parallel time of a scheduled DAG:

$$PT = \max_{n_i \in V}\{tlevel(n_i) + blevel(n_i)\}.$$

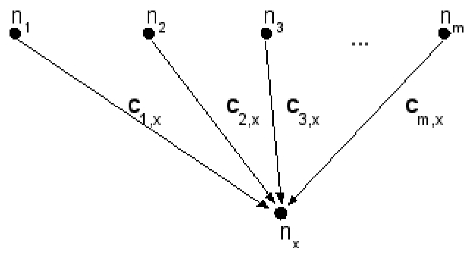
Table 4.2 summarizes some notations and their definitions.

4.2.3 Clustering of a DAG: communication and granularity

In presence of communication the scheduling problem is much harder [134, 54], mainly because the edge weights are no longer deterministic before the processor assignments have been done. Communication is zero when two tasks are mapped in the same processor, and nonzero when they are mapped in two different processors. Since the scheduling objective is the minimization of the makespan, assigning different tasks to the same processors can lead to a better schedule. The process of grouping different tasks is called clustering. Every task in a cluster must execute in the same



(a)



(b)

Figure 4.4: (a) Fork set. (b) Join set.

processor. Intuitively a graph is coarse-grained if the amount of computation is relatively large with respect to communication. Since with the clustering process we zero the communication edges by the assignment of the tasks to the same processor, clustering increases the grain of a DAG. Linear clustering groups tasks that are sequential in the original DAG, i.e. they belong to a simple directed path. Nonlinear clustering, instead, sequentializes parallel tasks and can improve the schedule length if communication is slow. So, a tradeoff must be found between parallelization (linear clustering) and sequentialization (nonlinear clustering). To study this problem a definition of *granularity of a DAG* must be given. Unfortunately in literature there is no consensus on this definition.

In [117] Stone studies the granularity by considering a task graph in which every task computes R units of time and communicates with all other tasks at a cost of C. He defines the *task granularity* as the ratio R/C and shows that this value determines the optimum tradeoff point between parallelization and sequentialization. In [52] Gerasoulis and Venugopal extend Stone's granularity definition for arbitrary DAGs. They introduce a quantity called *granularity of a DAG* defined as

$$g = \min_{x=1:v} \{\tau_x / \max_j \{c_{x,j}\}\}$$

where τ_x is the computation cost of node n_x and $c_{x,j}$ are the communication costs from node n_x to node n_j.

In [54] Gerasoulis and Yang study the determination of the tradeoff point between parallelization and sequentialization of tasks in arbitrary DAGs with the introduction of a different granularity definition for a DAG. With this granularity they prove that linear clustering is better than nonlinear clustering for arbitrary coarse grain DAGs. This is not true for other granularity definitions.

The building blocks of a DAG are the *fork* and *join* structures, as shown in Figure 4.4. In figure 4.4(a) is represented a fork primitive where the set $F_x = \{n_1, n_2, ..., n_n\}$ consists of all immediate successors of node n_x. Figure 4.4(b) shows a join primitive and the set $J_x = \{n_1, n_2, ..., n_m\}$ consists of all immediate predecessors of node n_x.

Given the following definitions:

$$g(F_x) = \min_{k=1:n}\{\tau_k\} / \max_{k=1:n}\{c_{x,k}\}$$

$$g(J_x) = \min_{k=1:m}\{\tau_k\} / \max_{k=1:m}\{c_{k,x}\}$$

where $n = |F_x|$ and $m = |J_x|$, the *grain* of a task is defined as:

$$g_x = \min\{g(F_x), g(J_x)\}.$$

Finally the expression for the *granularity* of a DAG is:

$$g(G) = \min_{x=1:v}\{g_x\}. \tag{4.1}$$

A DAG is *coarse grain* if $g(G) \geq 1$. For coarse grain DAGs each task receives or send a small amount of communication compared to the computation of its adjacent tasks. Equation 4.1 defines the granularity for arbitrary DAGs. Applying this definition to the class of DAGs studied by Stone [117], i.e. uniform computation times and uniform communication times, this granularity is equivalent to the one proposed by Stone. In fact, with $\tau_k = R$ and $c_{i,k} = C$, the equation 4.1 reduces to the ratio R/C.

With this definition of granularity, Gerasoulis and Yang [54] prove the following two theorems.

Theorem 4.2.1. *For any nonlinear clustering of a coarse grain DAG, there exists a linear clustering with less or equal parallel time.*

Theorem 4.2.2. *For any linear clustering algorithm we have:*

$$PT_{opt} \leq PT_{lc} \leq \Big(1 + \frac{1}{g(G)}\Big) PT_{opt}$$

where PT_{opt} is the optimum parallel time and PT_{lc} is the parallel time of the linear clustering.
Moreover for a coarse grain DAG we have:

$$PT_{lc} \leq 2 \times PT_{opt}.$$

Theorem 4.2.1 proves that for coarse grain DAGs the optimum clustering is a linear clustering, but without giving any performance bounds. This theorem is true only for the fork/join granularity definition 4.1. Theorem 4.2.2, instead, gives performance bounds for linear clustering and it is the one proposed by Gerasoulis and Venugopal [52]. Theorem 4.2.2 was originally formulated with another definition for granularity, but still holds for the granularity definition 4.1.

The bound given by theorem 4.2.2 relates to the result obtained by Graham [57]. He proposed a bound on the schedule length obtained by general list scheduling methods. Using a level-based approach for generating a list for scheduling, he found the following relation between the schedule length SL and the optimal schedule length SL_{opt}:

$$SL \leq \big(2 - \frac{1}{p}\big) SL_{opt}$$

In this case, however, communication edge costs are assumed to be either zero or additive to the task cost, Sarkar [107].

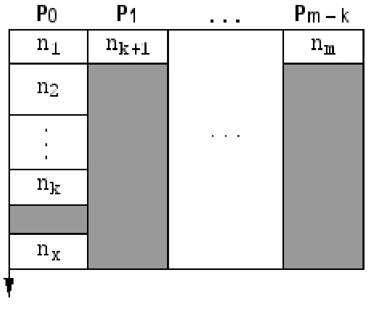
```
k = 1
while ($\sum_{j=1}^{k} \tau_j \leq \tau_k + c_{k,x}$) do
    $c_{k,x}$ = 0
    k = k+1
endwhile
k = k−1
```

Listing 4.3: Optimal algorithm for scheduling the primitive join structure.

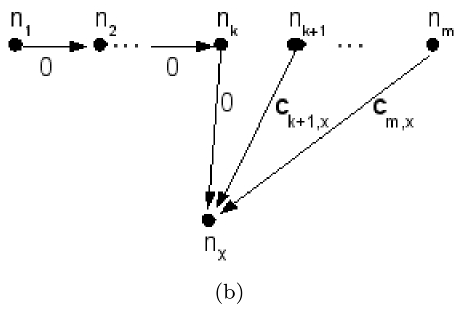
4.3 Scheduling of Primitive Graph Structures

The building blocks of a DAG are the fork and join components, also called primitive graph structures. Figure 4.4 shows an example of these two structures: the fork primitive in figure 4.4(a) and the join primitive in figure 4.4(b). In this section we report an algorithm for computing the optimal schedule of the fork and the join primitives in homogeneous computing systems. The study of the optimal schedule of the primitive graph structures gives some intuitive feeling on the meaning of the tradeoff between parallelization and sequentialization.

Now we consider the join set J_x of Figure 4.4(b). Without loss of generality, assume that for the join structure we have: $\tau_j + c_{j,x} \geq \tau_{j+1} + c_{j+1,x},\ j = 1..m-1$, i.e. the pairs $(\tau_j, c_{j,x})$ are sorted in descending order of their sum from left to right. The algorithm in listing 4.3 determines the optimum clustering for the join structure. At the beginning all tasks are mapped on separate processors. At each step k of the algorithm, the edge $e_{k,x}$ is visited and, if zeroing that edge reduces the parallel time, then this edge is zeroed. As can be seen from figure 4.5(b), zeroing an edge means sequentializing the corresponding task. Figure 4.5(a) shows the corresponding Gantt chart. Therefore, the task n_j for $j = 1..k$ will be mapped in the same cluster as long as the execution time for this cluster $\tau_x + \sum_{j=1}^{k} \tau_j$ remains less than $\tau_x + c_{k,x} + \tau_k$.



(a)



(b)

Figure 4.5: (a) Gantt chart and (b) scheduled DAG for the optimum clustering algorithm of the join primitive.

```
k = 1
while ($\sum_{j=1}^{k} \tau_j \leq \tau_k + c_{x,k}$) do
    $c_{x,k}$ = 0
    k = k+1
endwhile
k = k−1
```

Listing 4.4: Optimal algorithm for scheduling the primitive fork structure.

The algorithm will stop at some step k when the condition in the while loop becomes false, thus implying the following inequalities:

$$\sum_{j=1}^{k} \tau_j \leq c_{k,x} + \tau_k, \qquad \sum_{j=1}^{k+1} \tau_j > c_{k+1,x} + \tau_{k+1}.$$

Given such k satisfying the previous inequalities, the optimal schedule length for a join primitive is:

$$max\Big\{\tau_x + \sum_{j=1}^{k} \tau_j,\ \tau_x + c_{k+1,x} + \tau_{k+1}\Big\}$$

The optimum scheduling algorithm and schedule length for the *fork* primitive graph structure can be derived in an analogous way. Consider the fork set F_x of Figure 4.4(a) and assume that nodes and edges are sorted such that $c_{x,j} + \tau_j \geq c_{x,j+1} + \tau_{j+1},\ j = 1..n-1$. Listing 4.4 shows the algorithm that computes the optimal schedule for the fork primitive. It computes the tradeoff point k for which the following two inequalities hold:

$$\sum_{j=1}^{k} \tau_j \leq c_{x,k} + \tau_k, \qquad \sum_{j=1}^{k+1} \tau_j > c_{x,k+1} + \tau_{k+1}.$$

Finally, the optimal schedule length for the fork primitive is given by:

$$max\Big\{\tau_x + \sum_{j=1}^{k} \tau_j,\ \tau_x + c_{x,k+1} + \tau_{k+1}\Big\}$$

4.4 Taxonomy of DAG scheduling algorithms

This section outlines a taxonomy of scheduling algorithms for DAGs. A general taxonomy of scheduling in general-purpose distributed computing systems was presented by Casavant and Kuhl in [18]. Here we want to focus only on DAG scheduling algorithms, therefore this summary is by no means complete. We base our description on the taxonomy presented by Kwok and Ahmad [79, 80, 76]. Figure 4.6 shows a graphical description of the classification. The scheduling of parallel applications in

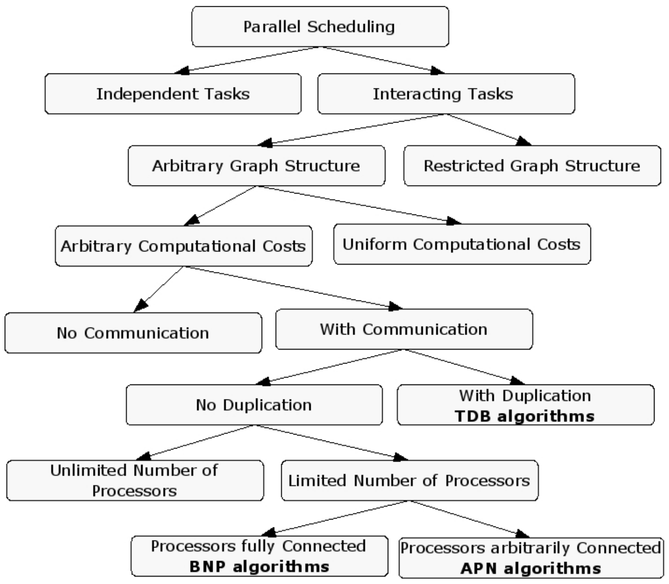

Parallel Scheduling
Independent Tasks
Interacting Tasks
Arbitrary Graph Structure
Restricted Graph Structure
Arbitrary Computational Costs
Uniform Computational Costs
No Communication
With Communication
No Duplication
With Duplication
TDB algorithms
Unlimited Number of Processors
Limited Number of Processors
Processors fully Connected
BNP algorithms
Processors arbitrarily Connected
APN algorithms

Figure 4.6: A taxonomy of the DAG scheduling problem.

a multiprocessor environment is divided into two main categories: scheduling of independent tasks and scheduling of not independent tasks. Independent tasks do not have precedence relations among each other and, therefore, they can start execution simultaneously [49, 55]. Also this part of the general scheduling problem was shown to be NP-complete [63]. In [13, 114, 12] some works describing the scheduling of independent tasks in a Grid environment can be found. Since we are interested in the other category of scheduling algorithms, we do not discuss further the scheduling of independent tasks. Earlier works on DAG scheduling have made simplifying assumptions about the model representing the parallel program and the interconnection network [22, 56]. Since we are dealing with an NP-complete problem, these simplifications, which assume special graph structures such as trees, fork-join, uniform computational costs, uniform communication costs or absence of communication, etc., were necessary in order to find polynomial solutions and for a better comprehension of the problem. In real life cases, however, parallel programs come in a variety of structures and as such many recent algorithms are designed to tackle arbitrary graphs and interconnection networks.

Scheduling with communication may be done with or without duplication. Task-duplication based (TDB) algorithms allow the duplication of tasks by redundantly allocating some nodes to multiple processors. The objective is to reduce the communication overhead and to increase the fault tolerance. Different strategies can be employed to select ancestor nodes for duplication. Some of the algorithms duplicate only the direct predecessors while others try to duplicate all possible ancestors [101, 2].

With respect to the multiprocessor model, some of the scheduling algorithms assume an unlimited availability of processors, while other scheduling algorithms assume an environment with a limited number of processors. The algorithms belonging to the former class are called UNC (*Unbounded Number of Clusters*) scheduling algorithms [67, 74, 78, 107, 131, 135, 76, 74], and the ones belonging to the latter class are called BNP (*Bounded Number of Processors*) scheduling algorithms [116, 98, 97, 93, 75, 66, 9, 1, 76, 74]. In both UNC and BNP the network is assumed to be fully-connected and no attention is paid to link contention or routing strategies used for communication. The technique employed by the UNC algorithms is also called clustering [53]. When the algorithm starts, each node is considered a cluster itself. In the subsequent steps, two clusters are merged if the merging reduces the completion time. This merging procedure continues until no cluster can be merged. One of the motivations behind the UNC algorithms is that they can take advantage of using more processors to further reduce the schedule length. However, the clusters generated by the UNC may need a post-processing step for mapping the clusters onto the processors because the number of processors available may be less than the number of clusters.

A few algorithms have been designed to take into account the most general model in which the system is assumed to consist of an arbitrary network topology, whose links are not contention-free. These algorithms are called APN (*arbitrary processor network*) and in addition to scheduling tasks they also schedule messages on the network communication links [3, 127, 115, 110, 103, 94, 73, 69, 62, 35, 32, 77].

4.5 Properties of list scheduling

Since the scheduling problem is NP-complete, considerable attention has been paid on the performance analysis of the heuristics used to solve this problem, e.g. Sarkar [107], El-Rewini and Lewis [35], Kim and Browne [67], Wu and Gajski [132], Consrad et al. [24], Darte [31], Yang and Gerasoulis [134] and many others. At an early stage of research the scheduling problem was simplified by assuming that communication cost is zero. In this context, Graham [57] proved that any list scheduling heuristic is within 50% of the optimum. Adam, Chandy and Dickson [1] demonstrated experimentally that the critical path (CP) list scheduling heuristic is within 5% of the optimum in 90% of the times. If communication is considered then list scheduling no longer provides the above mentioned performance characteristics.

In [134] Yang and Gerasoulis identify important properties of list scheduling heuristics. First they consider the case of list scheduling without communication and give a bound for the CP heuristic. We briefly summarize their result.

Consider the list scheduling as a sequence of *merging* operations. Initially all tasks are assigned to a unit cluster. At each step the selected task is left in its own cluster or merged to another cluster. Assume that PT^i is the parallel time at step i, then the merging sequence produces a sequence of parallel times $PT^0, PT^1, ..., PT^n$. This means that, after the last step n, the parallel time is: $PT = PT^n$.

Definition 4.5.1. *Let* PT^i_{opt} *be the optimal parallel time at step i, which is the solution of* $\min\{PT^i, PT^{i-1}\}$ *where the minimum is taken over all possible choices of* *free tasks at step i. Then a heuristic is called $\delta - opt$ if* $\max_i\{PT^i - PT^i_{opt}\} \leq \delta$ *where δ is a given constant. If $\delta = 0$ then the $\delta - opt$ heuristic is called a* local optimum.

They can prove the following theorem which gives an upper bound for δ for the CP heuristic.

Theorem 4.5.1. *The CP heuristic is $\delta - opt$ with $\delta \leq \Delta T = \max_{n_i, n_j \in V} |\tau_i - \tau_j|$.*

The important question is if local optimality implies near-optimality. It is very difficult to prove this result theoretically. Anyway, we do know that the CP heuristic is near optimal [1]. As a consequence of theorem 4.5.1 there is the following corollary, which proves that the CP heuristic is also locally optimal.

Corollary 4.5.1. *CP scheduling is locally optimal for DAGs with equal weights.*

In general, of course, local optimality does not imply optimality but a near optimal performance of heuristics that possess this property can be expected.

The scheduling problem is much harder if communication is considered. Yang and Gerasoulis extended their analysis in order to see what type of results can be obtained in the more interesting case of the list scheduling with communication. In this case, if a free, but not ready, task is scheduled it must wait for the data to arrive before it can start its execution and an idle gap is created in the schedule. On the other hand, if a ready task is chosen for scheduling then it can start its execution immediately. Therefore, there are two choices for list scheduling: Free List Scheduling (FLS) and Ready List Scheduling. In FLS the highest priority free task is selected for mapping, while in RLS the highest priority ready task is selected for mapping. Yang and Gerasoulis extended the $\delta - opt$ analysis to the ready list scheduling. RLS uses the CP heuristic, i.e. the ready tasks with highest priority are selected for mapping. Now the $\delta - opt$ definition is slightly different than the one given for list scheduling without communication. *The local optimum parallel time PT^i_{opt} is derived over all possible schedules of ready tasks for a selected processor.* Given this definition, the following theorem is true.

Theorem 4.5.2. *The RCP heuristic is $\delta - opt$ with $\delta \leq \Delta T = \max_{n_j, n_i \in V} |\tau_j - \tau_i|$.*

4.6 DAG scheduling algorithms

In this section we survey some of the DAG scheduling algorithms that can be found in the literature. We start with earlier optimal algorithms which assume a restricted DAG model and/or a restricted processor network model, e.g. unit computation weights and no communication. These are the Hu's algorithm [60] to schedule tree-structured task graphs with identical computation cost to an arbitrary number of processors, the Coffman and Graham [23] algorithm to schedule arbitrary task graphs with identical computation costs to two processors and the Papadimitriou and Yannakakis [99] algorithm to schedule an *interval-ordered* DAG with uniform node weights to an arbitrary number of processors.

We continue with the description of algorithms that were designed for homogeneous systems (Sarkar, HLFET, ETF, ISH, FLB, DSC, CASS-II, DCP, MCP and MD). Almost all the algorithms found in the literature work on homogeneous systems. Although the purpose of this thesis is to consider a heterogeneous and dynamic environment, studying these algorithms is very useful. The heuristics that they define can be used also in heterogeneous systems, at the cost of increasing the time-complexity of the algorithm.

We conclude with the Hybrid Remapper, an algorithm designed to work in heterogeneous systems.

4.6.1 Polynomial-time algorithm for Tree-Structured DAGs

This algorithm was proposed by Hu [60]. It is a linear time algorithm that assumes the following simplifications: the graph is an in-tree structured DAG, unit computations, without communication and a limited number of processors equal to p. Remember that an in-tree is an oriented tree in which a single vertex is reachable from every other one. Each node n_i is labeled with α_i, where $\alpha_i = x_i + 1$ and x_i is the length of the path from n_i to the exit node in the DAG. Since there is no communication and unit computational labels, the *length* x_i is the number of edges in the path. The algorithm constructs an optimal schedule for p processors by visiting the tree-structured graph in the following steps:

1. Schedule the first p (or fewer) nodes with the highest numbered label to the processors, i.e. at the beginning the algorithm schedules, at least, all the entry nodes of the DAG.

2. Remove the p scheduled tasks from the DAG and consider all the nodes without predecessors as entry nodes.

3. Repeat steps (1) and (2) until all nodes are scheduled.

In [60] Hu proves the optimality of the algorithm under the stated constraints.

4.6.2 Arbitrary graphs for a two-processor system

Coffman and Graham [23] proposed an algorithm for generating optimal schedules in case of arbitrary structured graphs with unit computation costs, no communication and a system with two processors. The first step of the algorithm is to assign labels (with increasing values) to the nodes. The graph is visited bottom-up in a way that considers as candidates for the assignment of the next label all the nodes whose successors have already been assigned a label. When this step is complete, the nodes are ordered by decreasing label numbers . The optimal schedule is computed by mapping ready tasks to idle processors. The algorithm is summarized in the following steps:

1. Choose one exit node and assign to it the label 1.

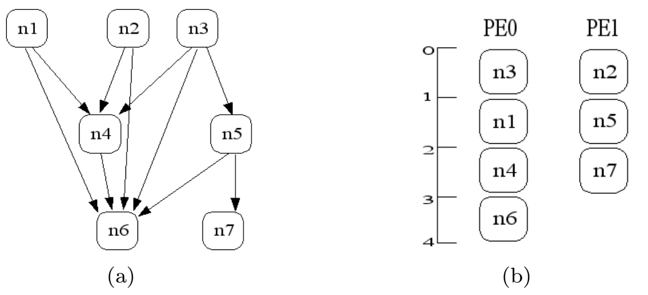


Figure 4.7: (a) An interval-ordered graph with unit computational costs. (b) An optimal schedule produced by the algorithm of Pappadimitrou and Yannakakis

2. At step j, labels $1, 2, ..., j-1$ have been assigned. Let S be the set of unassigned nodes with no unlabeled successors. The label j have to be assigned to an element of S as follows. For each node $x \in S$, consider the nodes $y_1^x, y_2^x, ..., y_k^x$ to be the immediate successors of x. Let $l(x)$ be the decreasing sequence of integers given by the set of y's labels. If $l(x) \leq l(x'), \forall x' \in S$, then assign the label j to the node x.

3. After the labeling process is complete, consider the list of tasks in descending order of labels and schedule each task to one of the two processors that allows the earlier execution of the task.

Using counter-examples, Coffmand and Graham demonstrated the their algorithm can generate sub-optimal schedules when the number of processors is increased to three or more or when the tasks are allowed to have arbitrary computation costs. Both the labeling process and the scheduling process take $O(v^2)$ time.

In [111] Sethi improved the complexity of the algorithm by the formulation of a $O(v + e)$ procedure to assign the labels and the use of data structures, for which v access operations can be performed in $O(v\alpha(v))$ (with $\alpha(v)$ be the inverse Ackermann's function), in the scheduling process. He observes that the labels of a set of nodes with the same height only depend on their children and he uses such information through visiting the edges connecting the nodes and their children, thus without constructing the lexicographic ordering. The final time complexity of the algorithm is $O(v\alpha(v)+e)$.

4.6.3 Scheduling of Interval-Ordered DAGs

A linear time algorithm that computes optimal solutions for scheduling a unit computational interval-ordered DAG to multiprocessors, was proposed by Pappadimitrou and Yannakakis [99]. In an interval-ordered DAG, two nodes are precedence-related if and only if the nodes can be mapped to non-overlapping intervals on the real line [38]. Given this property, the number of successors of a node can be used as a priority to construct a list. Figure 4.7 shows an example of interval-ordered graph. The optimal schedule can be constructed in $O(v+e)$. If the constraint of unit computational tasks is removed (allowing the tasks to have arbitrary computational weights), the problem becomes NP-complete.

A linear time algorithm to schedule interval-ordered graphs with unit computational costs and unit edge weights was proposed by Ali and El-Rewini [4].

4.6.4 Sarkar's algorithm

The first clustering algorithm that we consider is the Sarkar's algorithm [107], that is not a list scheduling algorithm. Initially it sorts the edges of the DAG in descending order of weights. Then at each clustering step the edges are visited in the sorted list order and zeroed if the parallel time does not increase. The computation of the parallel time must be done at each clustering step topologically traversing the scheduled DAG. Since there are e such traversals, the complexity is $O(e(v+e))$ (which dominates the $O(e\,log\,e)$ time necessary to sort the edges).

Gerasoulis and Yang [54] noted that imposing the linearity constraint the complexity reduces from $O(e(v+e))$ to $O(e\,log\,e)$. At each step the edges that will create nonlinear clusters, if they are zeroed, do not have to be visited. The linearity constraint ensures that the parallel time at each zeroing step does not increase and it must not be computed. Therefore, the cost for sorting the edges now dominates the time complexity of the algorithm. It is a BNP algorithm.

4.6.5 The HLFET algorithm

The HLFET (Highest Level First with Estimated Times) [1] is a list scheduling algorithm. It first computes the *static blevel* for each node and then makes a ready list in descending order of *static blevel* (ties are broken randomly). Then it repeatedly schedules the first node in the ready list to a processor that allows the earliest start time and updates the list with the new ready nodes. It is a BNP algorithm with $O(v^2)$ time complexity.

4.6.6 The ETF algorithm

In ETF (Earliest Time First) [61, 105] the main idea is to keep the processor busy, and in this respect being close to a load balancing scheme. At each scheduling step, the priorities for ready unmapped tasks are computed. The task priority is the earliest start time, which is determined by tentatively mapping the given task to all the processors. The selected task is the one with the lowest priority and the selected processor is the one corresponding to this priority. Ties are broken considering statically computed priorities. The time complexity of the algorithm is $O(v^2p)$. This is not a critical path based heuristic and the algorithm does not always map the most important ready task first. Therefore, it may not be able to reduce the partial schedule length at every scheduling step. It is a BNP algorithm, since the earliest start time of a node is computed by examining the start-time of a node on all processors exhaustively.

4.6.7 The ISH algorithm

The ISH (Insertion Scheduling Heuristic) algorithm [71, 80] tries to fill, as much as possible, the idle time slots of the processors. It selects ready nodes giving priority to the ones with higher *blevel* value (the *blevels* of the nodes are computed statically at the beginning of the algorithm). The selected node is assigned to the processor that allows the earliest execution, using the non-insertion algorithm (the task is scheduled to the processor using a FIFO queue). If the scheduling of this node causes an idle time slot, then the algorithm schedules as many nodes as possible into this idle time slot, providing that these nodes cannot be scheduled earlier on other processors. Then it selects a new ready node and repeats all these steps. The time complexity of the ISH algorithm is $O(v^2)$. It is a BNP algorithm.

4.6.8 The FLB algorithm

The FLB (Fast Load Balancing) algorithm [104] is an improvement of the ETF algorithm. It uses the same task selection criteria but the preferred task is identified in $O(log\ p)$ instead of $O(wp)$, where w is the *width* of the DAG and p the number of processors. The **width** of a DAG is the maximum number of independent tasks. Although essentially similar, there is a small difference in the task selection scheme. Throughout the scheduling process it may happen that several ready tasks can start at the same earliest time. ETF and FLB have different criteria to break this tie. It is a BNP algorithm.

4.6.9 The DSC algorithm

The Sarkar's algorithm described above, zeroes the highest communication edge. The edge, however, might not be in a DS that determines the parallel time and as a result the parallel time might not be reduced at all. The main idea behind the DSC (Dominant Sequence Clustering) algorithm is to perform a sequence of edge zeroing steps with the goal of reducing a DS at each step [135].

The initial step assumes that each node is mapped in a unit cluster, then the algorithm tries to merge appropriate clusters by zeroing the edge which connects them. During the initialization step the *blevel* is computed for each node and the *tlevel* is computed for each *free* node. Two lists are initialized: a free task list (FL) and a partially free task list (PFL). The free task list is ordered in descending order considering this priority definition:

$$PRIO(n_x) = tlevel(n_x) + blevel(n_x).$$

Hence, a DS node can be identified as the one with the highest priority. The *tlevel* of a partially free task is defined as:

$$ptlevel(n_y) = \max_{n_j \in PRED(n_y) \cap EG} \{tlevel(n_j) + \tau_j + c_{j,y}\}$$

where EG is the set of examined nodes. The priority for the nodes in the partially free task list can be defined as:

$$pPRIO(n_y) = ptlevel(n_y) + blevel(n_y).$$

The priority of a partially free node n_y is computed considering only the examined parents of n_y. The subsequent steps of the algorithm are briefly described below:

1. Initialization:
    - Add all free entry nodes to FL.
    - Compute *blevel* for each node.
    - Set $tlevel = 0$ for each free node.
2. $n_x = head(FL)$ and $n_y = head(PFL)$.
3. If $PRIO(n_x) \geq pPRIO(n_y)$ then minimize $tlevel(n_x)$ else minimize $tlevel(n_x)$ under constraint DSRW.
4. Update the priorities of n_x's successors.
5. Repeat steps 1-3 until all nodes are examined.

Minimization procedure To reduce $tlevel(n_x)$ the minimization procedure zeroes multiple incoming edges of the free task n_x. If no zeroing results in a lowering of the *tlevel* then the node remains in a single node cluster. Major details can be found in [135]. The cost of the minimization algorithm is $O(m \; log \; m)$, where m is the number of predecessors of n_x.

Dominant Sequence Length Reduction Warranty (DSRW) The DSRW constraint is applied when there is no DS going through any free task and there is one DS passing through a partially free node n_y. In this case zeroing non-DS incoming edges of free nodes could affect the reduction of $tlevel(n_y)$ in the future steps. Thus, the DSRW states that zeroing edges of a free node should not affect the future reduction of the $tlevel(n_y)$, if it is reducible by zeroing an incoming DS edge of n_y. If n_p is one examined predecessor of n_y and zeroing $e_{p,y}$ would reduce $ptlevel(n_y)$, then the DSRW constraint is implemented by imposing that no other nodes are allowed to be mapped to the resource assigned to n_p until n_y becomes free.

The time complexity of the DSC algorithm is $O((e + v) log \, v)$. It is an UNC algorithm.

4.6.10 The CASS-II algorithm

The CASS-II (Clustering And Scheduling System II) algorithm [85] is analogous to the DSC algorithm with the difference that it constructs the clusters bottom up, i.e., starting from the sink nodes. It employs a two step approach. In the first step, CASS-II computes for each node its *tlevel*. These values are computed in topological order of the graph.

The second step is the clustering step. First the algorithm computes the *blevel* for each node. The algorithm begins by placing every sink node n_x in its own cluster, and by setting $blevel(n_x) = \tau_x$. The algorithm then goes through a sequence of iterations, where at each iteration it considers for clustering every node n_y whose immediate successors have been clustered (call such a node *current*). For each n_y, $blevel(n_y)$ can be easily computed since all the successors of n_y are clustered and hence been assigned *blevels*. The *current* nodes are considered for clustering in non-increasing order of their $tlevel + blevel$. For a given current node n_y, let n_x be its dominant successor (the one that determines $blevel(n_y)$) and let C_x be the cluster containing n_x. Then, n_y is included in the cluster C_x if doing so does not increase both $blevel(n_y)$ and $blevel(C_x)$ (the maximum of all the *blevels* of the nodes belonging to C_x). Otherwise, n_y is placed in a new cluster.

It is an UNC algorithm.

4.6.11 The DCP algorithm

The DCP (Dynamic Critical Path) algorithm [78][80] is based on the mobility attribute, the difference between ALAP (As Late As Possible, the authors use the term ALST, i.e. Absolute Latest Start Time) and ASAP (As Soon As Possible, the authors use the term AEST, i.e. Absolute Earliest Start Time). The mobility attribute for a node n_x can be defined as: $DS - \big(blevel(n_i) + tlevel(n_i)\big)$. Nodes with lower mobility are selected first. The minimum mobility value is zero and a node with zero mobility belongs to the CP. Then, the DCP algorithm uses a *lookahead* strategy to find a better cluster for a given node. The algorithm is described below:

1. Compute ALAP and ASAP for all nodes.
2. Select the node n_x with lower mobility. Let n_c be the unscheduled child of n_x with the largest communication.
3. Select a processor P such that ASAP(n_x)+ASAP(n_c) is the smallest among all processors holding n_x's parents or children. In examining a processor, first try to find an idle time slot. If this is unsuccessful try to create an idle time slot pulling some already scheduled nodes downward (considering the mobility in order to not increase the schedule length). In case of failure select a new processor.
4. Schedule n_x to P.
5. Update ALAP and ASAP for all nodes.
6. Repeat 2-4 until all nodes are scheduled.

The *lookahead* strategy aims to avoid scheduling a node to a processor that has no room to accommodate a heavily communicated child of the node. All attributes are computed dynamically, that means that they are processor dependent and updated after each step of the algorithm. The time complexity of DCP is $O(v^3)$.

4.6.12 The MCP algorithm

The MCP (Modified Critical Path) algorithm [133] selects a node using the ALAP (As Late As Possible) time as a priority, allowing to find nodes belonging to the CP. It first computes the ALAP times of all the nodes and then constructs a list of nodes in ascending order of ALAP times (ties are broken by considering the ALAP times of the children f a node). Finally, the nodes on the list are scheduled one by one such that the selected processor is the one that allows the earliest start time using the insertion approach. The time complexity of the MCP algorithm is $O(v^2 \log v)$. It is an UNC algorithm.

4.6.13 The MD algorithm

The MD (Mobility Directed) algorithm [133][80] defines a priority criteria called *relative mobility*, which is defined as:

$$\frac{DS - \big(blevel(n_x) + tlevel(n_x)\big)}{\tau_x}$$

where τ_x is the execution time of task x. If a node is on the current CP of the partially scheduled DAG, the sum of its *blevel* and *tlevel* is equal to the current CP length and, therefore, the relative mobility of a CP node is zero.

The algorithm works like this: at each step, the MD algorithm schedules the node with the smallest mobility to the first processor which has a large enough time slot (eventually pulling already scheduled nodes downward in a way that respects their mobility) to accommodate the node, but without considering the minimization of the node's start-time. This processor selection procedure has an evident drawback, it can increase the schedule length. The time complexity of the algorithm is $O(v^3)$. It is an UNC algorithm.

4.6.14 The Hybrid Remapper algorithm

The Hybrid Remapper [90] is a dynamic list scheduling algorithm specifically designed for heterogeneous environments. Like the other algorithms its starting point is an initial DAG labeled with execution and data transfer times (obtained by an initial static matching and scheduling). Then, the algorithm executes in two phases. In the first phase the set of tasks is partitioned into blocks such that the tasks in a block do not have any data dependencies among them. However, the order among the blocks is determined by the data dependencies that are present among the tasks of the entire DAG (it is a kind of level decomposition). In particular, all tasks that send data to a child task in block k must be in any of blocks 0 to $k-1$ and for each task in block k there exists at least one incident edge such that the corresponding parent task is in block $k-1$. Once the tasks in the DAG are partitioned, each task is assigned a rank by examining the tasks from block $B-1$ to block 0, where B is the number of blocks. The rank is computed as follows:

$$rank(n_x) = \tau_{x,i} + \max_{n_y \in iss(n_x)} \{c_{x,y} + rank(n_y)\} \tag{4.2}$$

where $\tau_{x,i}$ is the expected computation time of the task n_x on machine m_i and $c_{x,y}$ is the data transfer time from n_x to n_y. The specification of the machine in the notation of the expected computation time is needed because the algorithm is considering a heterogeneous environment. The rank is equivalent to the *blevel*. The second phase is executed during application run-time and involves remapping the tasks considering changes to the initial informations (statically determined).

The hybrid remapper algorithm is presented in three variants. In all the three versions the execution of the tasks proceed from block 0 to block $B-1$. The hybrid remapper starts examining the block k tasks when the first block $(k-1)$ task begins its execution. There may be some tasks from blocks 0 to $k-2$ that are still running or waiting execution when tasks from block k are being considered for remapping. For such tasks, expected completion times are used.

Version 1. Minimum Partial Completion Time Static Priority (PS). The priority of a node n_x is equal to the *blevel(n_x)* computed statically in the first phase. Node's selections proceed considering high priority nodes. The matching criterion used is the minimization of the partial completion time. The partial completion time of a node n_x is essentially the expected finishing time of the node in the considered machine. Let $ips(n_x)$ be the immediate predecessor set for task n_x and $n_y \in ips(n_x)$ is mapped onto machine m_j. For any task n_x with no incoming edges mapped in machine m_i, $pct(n_x, i) = \tau_{x,i}$. $dr(n_x)$ is the time at which the last data item required by n_x arrives at m_i, then the partial completion time of task n_x mapped onto machine m_i is defined as:

$$dr(n_x) = \max_{n_y \in ips(n_x)} \{c_{y,x} + pct(n_y, j)\}$$

$$pct(n_x, i) = \tau_{x,i} + max\{A[i], dr(n_x)\}$$

where $A[i]$ is the time at which the machine m_i is available. This definition considers changes in data transfer time from predecessors, while choosing the best machine for the task.

Version 2. Minimum Completion Time Static Priority (CS). This variant maintains the same criteria as the previous one (PS) for node's selection. The difference lies in the matching criteria. Now it attempts to minimize the overall completion time by remapping each task n_x in block k such that the length of the critical path through task n_x is reduced. The PS variant is faster but CS attempts to derive a better mapping because it considers the whole critical path through n_x. Machine m_i is selected for task n_x if it minimizes the completion time $ct(n_x, i)$, and $A[i]$ is updated using $pct(n_x, i)$. The completion time $ct(n_x, i)$ is the weight of the critical path passing through n_x which is mapped on m_i.

$$\begin{aligned} ct(n_x, i) &= \max_{n_y \in iss(n_x)} \{pct(n_x, i) + c_{x,y} + rank(n_y)\} \\ &= pct(n_x, i) + \max_{n_y \in iss(n_x)} \{c_{x,y} + rank(n_y)\} \end{aligned}$$

This time changes in transfer times with n_x and its children are considered.

Version 3. Minimum Completion Time Dynamic Priority (CD). In the previous two versions the priority was computed statically, in particular the rank of a task n_x is computed prior to application execution. This version considers the same matching criteria as the one used in the CS version, but it introduces a dynamic priority for node selection. A node n_x is selected from block k if $ct(n_x, i)$ is the greatest among all block k nodes (this time the selected n_x is in the CP). m_i is the machine assigned to n_x in the initial mapping.

4.7 Summary

In this chapter we have presented background and a brief survey of the most representative DAG scheduling algorithms found in the literature and considered both the static and the dynamic DAG scheduling problem. The DAG describes the parallel application that have to be scheduled to the target multiprocessor system. A node represents a task which is a set of instructions that must be executed sequentially in the same processor. An edge denotes the communication and data dependency between two program tasks. Nodes and edges are labeled with values representing the execution time and the data transfer time respectively. The computing system, which executes the parallel application, is generally assumed to be a network of processing elements (PEs). There can be two types of computing system: homogeneous and heterogeneous. In heterogeneous systems usually the communication links are assumed to be homogeneous, i.e. a message is transmitted with the same speed on all links. In Grid this assumption cannot be made and a heterogeneous interconnection network must be considered. The objective of scheduling is to minimize the schedule length by properly allocating the nodes to the computing system without violating precedence constraints.

We have presented a brief report of the NP-completeness results of various simplified variants of the problem, thereby illustrating that scheduling is a hard optimization problem. Then we have showed how the problem becomes even more complicated when communication is taken into consideration. In this case the DAG scheduling problem consists in finding the tradeoff between parallelization and sequentialization. Depending on its granularity, which is a measure of the communication to computation ratio, a DAG can be coarse grained (the computation dominates the communication) or fine grained (the communication dominates the computation).

As the problem is intractable even for moderately general cases, heuristic approaches are commonly sought. The most popular implementation schema is the list scheduling approach: higher priority nodes are selected first and mapped to a suitable resource. With this technique the task graph structure is carefully exploited to determine the relative importance of the nodes in the graph. More important nodes get a higher consideration priority for scheduling first. Experimentally, priority heuristics based on the Critical Path are the ones that give better results.

The chapter concludes with a description of the most representative DAG scheduling algorithms. All the considered algorithms accept a general DAG. Whereas with respect to the computing system some of them assume a homogeneous system, others a heterogeneous system with homogeneous interconnection network and others a heterogeneous system with a heterogeneous interconnection network.

5
A DAG scheduling algorithm for Grid computing systems

Today's parallel and distributed systems are changing in their organization and the concept of Grid computing, a set of dynamic and heterogeneous resources connected via Internet and shared by many users, is nowadays becoming a reality. A large number of scheduling heuristics for parallel applications described by directed acyclic graphs (DAGs) have been presented in the literature, but most of them assume a homogeneous network, i.e. a message is transmitted with the same speed on all the links. In a Grid environment this assumption cannot be done. In this chapter we tackle the problem of scheduling directed acyclic dags in metacomputing or Grid systems, i.e. heterogeneous systems with a heterogeneous interconnection network. We present two dynamic task scheduling algorithms: the first one is called CCF, it is a list scheduling algorithm and uses a two phase selection of nodes giving priority to ready children of a task; the second one is a variant of the well known DSC algorithm.

5.1 The computing system environment

We have seen in the previous chapter that the DAG scheduling problem is NP-complete in its general formulation and that polynomial time optimal solutions are known only for special cases assuming simplifying conditions. These simplifying conditions regards the DAG structure and the fact that communication is (mostly) not taken into account. Due to the intractability of the general scheduling problem, many heuristics have been suggested to tackle the problem under more generic situations. A heuristic produces an answer in less than exponential time but does not guarantee an optimal solution. A heuristic is said to be better than another heuristic if solutions approach optimality more often, or if a near-optimal solution is obtained in less time. The effectiveness of these scheduling heuristics depends, in most cases, on both the parallel application and the target machine.

In the literature many algorithms working on homogeneous and heterogeneous systems can be found. However, many of the works, even considering communication and a heterogeneous environment, assume a homogeneous interconnection network, i.e. all the links have the same latency and bandwidth and, therefore, messages are transmitted with the same speed on all the links. In the design of a scheduling algorithm for the Grid, simplifying assumptions concerning the computing system environment cannot be made: all the resources must be heterogeneous, both the machines and the network. In fact, the Grid consists of heterogeneous workstations connected by local-area networks (LANs) and/or wide-area networks (WANs). Moreover, Grid is a dynamic environment where resources come and go and are shared by many different users. Given these assumptions, the scheduler have to be dynamic and flexible, i.e. schedules have to be computed using updated values and a regular monitoring of the system is necessary in order to change the computed schedule if the external conditions have significantly changed. Such a kind of heterogeneous and dynamic environment leads to relevant consequences on the design of a scheduler, in particular from a time complexity point of view. For example, when we want to find the best resource for a task we have to “test” this task on each machine belonging to a set of candidate resources. Every time a resource is considered, estimates on task execution time, together with estimates on incoming and outgoing data transfer times, must be computed. In a homogeneous system this step can be done only once at the beginning and in a heterogeneous system with a homogeneous interconnection network only the estimates on task execution times have to be computed, data transfer times do not depend upon the source and the target machines. Fortunately, in a Grid-like system where tasks usually take more than few seconds to execute and the middleware overhead is significant, if the scheduler is designed to run in parallel with the application, i.e. computing the schedule in an incremental way, time constraints to compute the schedule of the application can be relaxed.

Grid computing technologies enable wide-spread sharing and coordinated use of networked resources. Sharing relationships may be static and long-lived, e.g. among the major resource centers of a company or university, or highly dynamic, e.g. among the evolving membership of a scientific collaboration. In either case, the fact that users typically have little or no knowledge of the resources contributed by participants in the “virtual organization” (VO) poses a significant obstacle to their use. For this reason, information services designed to support the initial discovery and ongoing monitoring of the existence and characteristics of resources, services, computations, and other entities are a vital part of a Grid system. The Globus Monitoring and Discovery Service [26] is an example of Grid service that allows the monitoring and the discovery of the resources. Another example is the Network Weather System [130] that is a distributed monitoring system designed to track and forecast dynamic resource conditions, e.g. it allows a user or a program (such as a scheduler) to request information (latency, bandwidth, load, estimates, etc) for entities corresponding to network links connecting specified endpoints, retrieve the fraction of CPU available to a newly started process, retrieve the amount of memory that is currently unused in a remote host, etc. These services can be used by the scheduler in order to discover the resources and to compute the estimates of task execution times and data transfer times.

5.2 Considerations on the design of the algorithm

A Grid scheduler can use the GIS (Grid Information System) to discover the resources and to find specific information about these resources. Grid scheduling involves three main phases [109]:

resource discovery: generation of a set of potential resources;

system selection: determines from the possibly large set of resources obtained in the previous phase the best set of resources, gathering detailed dynamic information;

job execution: once the task is mapped to a resource it is submitted for execution.

The parallel application is initially described by the abstract DAG (section 4.2) and the derivation of the concrete DAG should be done during or after the first two phases reported above. In this step there is no submission of tasks to the resources. The concrete DAG is the starting point of the scheduling algorithm.

We characterize the proposed algorithm as an *application-level* scheduler [13]. More generally, if a programmer wants to take advantage of Computational Grids, they are responsible for all transactions that require knowledge of the application at hand and, while many scientists could benefit from the extensive resources offered by Computational Grids, application development remains a daunting proposition. One solution is to develop software that frees the user of these responsibilities by providing an integrated Grid application development solution that incorporates activities such as compilation, scheduling, staging of binaries and data, application launch, and monitoring of application progress during execution. One example of a project finalized to the realization of this objective is the Grid Application Development Software (GrADS) project [11, 29]. An **application-level** scheduler is a scheduling algorithm integrated into the application, it can use low-level application specifications determined at compile-time in order to compute time estimates once the characteristics of the resouces are know at run-time. In the GrADS project these information are called performance models [25, 28]. Performance models are an architecture-independent model of the workflow component providing an estimated resource usage that can be converted to a rough time estimate when architectural parameters of the resource will be available. Both the number of cpu operations executed and the memory access pattern are considered for building up the component model.

There are also other possibilities than the application-level approach. In a more general Grid environment context a scheduler can be part of a RMS and in this case the structure of the scheduler depends on the number of resources on which jobs and computations are scheduled, and the domain in which resources are located. In this context there are three different models for structuring schedulers: centralized, decentralized and hierarchical [14]. We do not consider further the general characterization of the scheduler since our intention is to focus on the scheduling algorithm, that is rather independent from the context in which it is inserted. Obviously, if the scheduler is not integrated into the application and performance characteristics are needed by the scheduling algorithm, then the performance model of the application must be given as input.

5.3 Preliminaries

In this section we briefly recall and define some basic concepts and terms. We start with the description of the environment. The considered system is composed by heterogeneous workstations connected by local-area networks (LANs) and/or wide-area networks (WANs). Let M be the set of resources, which we refer to also as machines, hosts or processors. $|M|$ is the total number of machines and h_i denotes the *i-th* host. $\tau_{x,i}$ is the expected computation time of task n_x on host h_i. The earliest time at which host h_i is available is given by $aval(h_i)$, which returns zero if the machine is free (immediate available). With h_{n_x} we denote the host assigned to the task n_x.

The parallel application to be scheduled is described by a DAG (defined in section 4.2). We briefly recall some notation: let $c_{x,y}$ be the communication cost incurred along the edge $e_{x,y} = (n_x, n_y) \in E$, which becomes zero if both nodes are mapped to the same processor. The values $c_{x,y}$ and $\tau_{x,i}$ are estimates computed on the given hosts h_{n_x} and h_{n_y}. In order to compute these values the scheduling algorithm must know performance characteristics of the parallel application (e.g. amount of operations for the tasks and size of data to be transferred between two tasks) and architectural characteristics of machines and links (like cpu speed, links latency and bandwidth, loads, etc.). The latter can be retrieved using the GIS.

The objective of the scheduling process is to minimize the schedule length or *makespan* without violating precedence constraints.

5.4 The CCF algorithm

5.4.1 Considerations

It is evident from the literature that algorithms based on CP heuristics are the ones giving, on average, the best results in terms of quality of the schedule produced. However, while the CP length provides a lower bound on the schedule length, making all the nodes on the CP start at the earliest possible time (or finish the execution as soon as possible in case of heterogeneous environments) does not guarantee an optimal schedule. One motivation of this fact is that in the course of scheduling, the CP can change and the set of nodes constituting the CP at a certain scheduling step may be different from that at an earlier step. Indeed, the algorithms that dynamically select nodes for scheduling such as the MD, DLS, DSC and DCP algorithms (discussed in Chapter 4), are designed based on this observation. They consider the nodes belonging to the DS, therefore tracking dynamical changes in the CP. Unfortunately, as we have seen, the general DAG scheduling problem is NP-complete and even these dynamic heuristics can get trapped in a locally optimal decision, leading to a non-optimal global solution. This means that scheduling at each step a DS node may not be the correct choice.

The proposed CCF (Cluster ready Children First) algorithm belongs to the class of the dynamic list scheduling algorithms. In particular, in order to keep track of

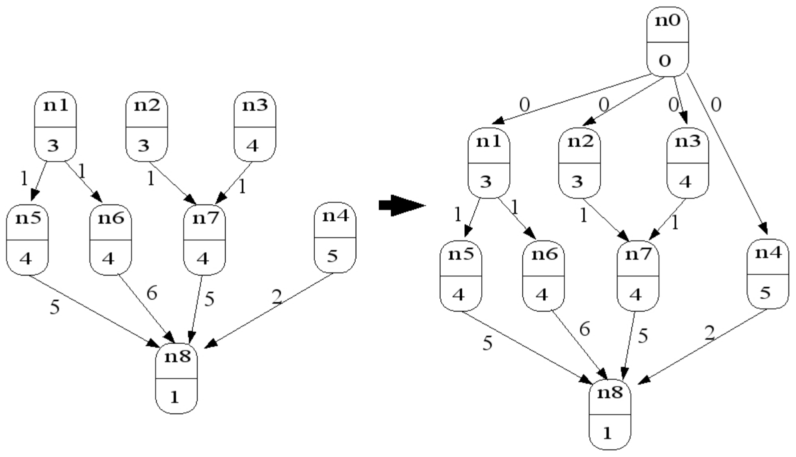


Figure 5.1: The conversion from a general DAG to an equivalent DAG with one entry node is done by adding one entry node with computation time set to zero and connecting, with zero communication cost links, all the nodes of the original DAG with no incoming edges.

application execution and environment changes, the DAG is visited in topological order and tasks are submitted as soon as definitive scheduling decisions are taken for the considered tasks. The algorithm runs in parallel with the application execution. It follows the schema of the list scheduling approach: the task with the highest priority is selected and then it is mapped to the most suitable resource. The algorithm is based on a heuristic that tries to schedule nodes that are "near", but not necessarily on, the DS. All the details are given in the following sections.

5.4.2 Assumptions

Without loss of generality we consider DAGs with one entry node. A general DAG can be transformed in an equivalent DAG of that form by adding one entry node with computation time set to zero and connecting, with zero communication cost links, all the nodes with no incoming edges (figure 5.1). This is a realistic assumption since the first node can be typically identified with the user machine.

DAG scheduling algorithms usually assume, as starting point, a concrete DAG. Therefore, the proposed algorithms runs after an initial static matching and scheduling phase that produces the concrete DAG.

We assume the availability of a GIS (Grid Information System) that provides services for the discovery and monitoring of the resources. These services are used by the initial mapping phase for finding the set of resources to be assigned to the tasks and for the definition of the concrete DAG. More specifically, these services, together with the performance model of the application, are necessary in order to obtain the

```
Compute tlevels and blevels
Insert sourceTask into RUNNING_QUEUE
while (RUNNING_QUEUE is not empty) do
  task = extract(RUNNING_QUEUE)
  foreach child of task do
    Insert child into CHILD_QUEUE
  endfor
  while (CHILD_QUEUE is not empty) do
    childTask = extract(CHILD_QUEUE)
    if (childTask is ready) then
      assignResource(childTask)
      updateSuggestedResource(childTask)
      Insert childTask into RUNNING_QUEUE
    else
      suggestResource(childTask)
    endif
  endwhile
endwhile
```

Listing 5.1: Outline of the CCF algorithm.

estimates of task execution times on a given host ($\tau_{x,i}$) and of data transfer times between source and target hosts ($c_{s,t}$).

5.4.3 First version of the algorithm

We are going to present the algorithm in two steps, we give a first initial version and perform some analysis on its behavior on the primitive graph structures *fork* and *join*. The final version of the algorithm will take into consideration the results of this analysis.

Given an initial mapping, CCF is a dynamic list scheduling algorithm that assigns, at run-time, resources to tasks described by a DAG. It is run-time and dynamic because the algorithm executes in parallel with the application and the information on which scheduling decisions are taken are regularly updated. Following the list scheduling schema, the CCF algorithm selects tasks considering a particular priority definition and assigns resources according to some cost minimization criteria. The algorithm is outlined in listing 5.1. The DAG is visited in topological order and a two phase task priority selection is implemented. When a task is assigned to a resource it is submitted for execution and inserted into the *RunningQueue*. The entry task of the DAG is assumed to be statically mapped (for example to the user machine) and it is automatically inserted into that queue at the beginning of the algorithm.

The first selection phase is based on the extraction of tasks from the *RunningQueue*, i.e., on the selection of tasks already mapped to a resource. The priority used by the

RunningQueue is defined as:

$$Prio\_RQ(n_x) = \max\Big\{tlevel(n_x, h_{n_x}) + blevel(n_x, h_{n_x}), DS_{desc}(n_x)\Big\} \tag{5.1}$$

where h_{n_x} is the host assigned to the task n_x and $DS_{desc}(n_x)$ is the value of the longest path passing through a partially free descendant of n_x. The value DS_{desc} is set by the *suggestResource*() procedure and will be discussed later. The extract operation returns (and removes from the queue) the task with the highest priority value, which is the node belonging to the *Dominant Sequence* if the priority value is not given by $DS_{desc}(n_x)$, otherwise it selects a node having a partially free DS node as descendant. Once a task is extracted from that queue all its children are inserted into the *ChildQueue*.

The second selection phase empties this queue and assigns a resource for each ready child. The *ChildQueue* orders the tasks using the following priority:

$$Prio\_CQ(n_y) = tlevel(n_y, h_{n_y}) + blevel(n_y, h_{n_y}) \tag{5.2}$$

where h_{n_y} is the host assigned to n_y by the initial mapping. The *while* loop in line 8 (listing 5.1) considers each child n_y and checks if it is ready or not (i.e. all its parents are mapped or not). In the first case (lines 11-13) a resource is assigned to the task and it is inserted into the *RunningQueue*. The function *updateSuggestedResouce*() in line 12 will be explained later. If the task is not ready (line 15), this means that some of its parents are not mapped and the procedure *suggestResource*() suggests a resource (h_{n_x}, the resource assigned to the father n_x extracted in line 4) to all the unmapped parents.

Following this schema, children with one parent are immediately mapped to an appropriate resource, whereas children with more than one parent wait until all parents are scheduled. It should be noted that the first selection phase considers high priority tasks in the *RunningQueue*, i.e. tasks with an already assigned resource that are, potentially, running. It is the second selection phase that looks for ready tasks needing a resource. This approach is different from all the others used in list scheduling algorithms. In particular, the priority definition (5.1) needs some more explanation. The main objective is to select DS nodes. However, if the heaviest child n_h of a selected DS node is not ready, then it is not assigned to a resource and waits until all its parents are mapped. Therefore, it is not guaranteed that the next selected node from the *RunningQueue* will be a DS node, unless there is more than one DS. The idea is to stay “near” the DS by making n_h ready as soon as possible. This happens when a task is selected from the *RunningQueue* because the value DS_{desc} is the maximum. DS_{desc} is initially set to zero for all the tasks and is possibly updated by the *suggestResource*() procedure, which is discussed later. If n_h is not ready the second selection phase can choose non-DS tasks for mapping. This makes this heuristic a hybrid between CP-based and non-CP-based heuristic. The next paragraphs describe the mapping function *assignResource()* and the functions *suggestResource*() and *updateSuggestedResouce*().

The mapping function *assignResource()*

This function assigns a suitable resource to a task. There are two important aspects: the set of candidate resources and the cost function to minimize.

The set of the candidate resources C is composed by:

- the resource assigned by the initial static mapping to the task;
- the resources of the task parents;
- one suggested resource.

The suggested resource, if present, is defined by the functions *suggestResource*() and *updateSuggestedResource*. The resource assigned by the initial static mapping is the reference resource, i.e. the one that gives the cost that have to be minimized. The cost function of the task n_y on a given host h, is defined as:

$$Cost(n_y, h) = tlevel(n_y, h) + blevel(n_y, h).$$

The function *assignResource*() selects the resource that minimize this cost function, by testing n_y on all the resources h_i belonging to the set of candidate resources. When the task n_y is tested on the machine h_i the cost function $Cost(n_y, h_i)$ is computed as follows:

1. the estimate of execution time of task n_y on the host h_i is computed ($\tau_{y,i}$);
2. if a predecessor n_p is assigned to h_i the edge (n_p, n_y) is zeroed, otherwise the estimate of the data transfer time $c_{p,y}$ is recomputed.
3. if a successor n_s is assigned (by the initial mapping) to h_i the edge (n_y, n_s) is zeroed, otherwise the estimate of the data transfer time $c_{y,s}$ is recomputed.

For selected resource h_{n_y} holds:

$$Cost(n_y, h_{n_y}) = \min_{h_j \in C} \Big\{ Cost(n_y, h_j) \Big\}.$$

The *suggestResource()* and *updateSuggestedResource()* functions

The rationale behind the suggested resource is to increase the probability of zeroing multiple incoming edges. Obviously, in order for a task to zero multiple incoming edges when assigned to a machine, there must be more than one of its parents assigned to the same machine. In figure 5.2 task n_1 is selected from the *RunningQueue* and task n_6 is its heaviest child (the one laying on the CP, indicated by the thick arrow). n_6 is not ready and it will be considered for scheduling only when all its parents are definitively assigned to a resource. At the moment of scheduling the task n_6, there could be an advantage if tasks n_3, n_4 and n_5 are also assigned to h_{n_1}. In some way, tasks n_2, n_3, n_4 and n_5 should be aware that they are converging towards one task (n_6 in this case).

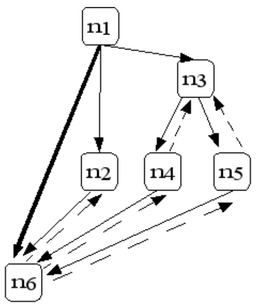


Figure 5.2: Task n_6 suggests the resource of task assigned to n_1 to all its ascendants.

So, when a not ready child n_y is extracted from the *ChildQueue* the scheduling decision is delayed for as long as all its parents are mapped to a resource. The suggested resource tries to influence the selection of a resource for all the ascendants of n_y, i.e. if more fathers of n_y are clustered in the same resource then more incoming edges of n_y can be zeroed choosing that resource. This method tries to exploit the information associated to the early visit of a not ready task. At the time of the first visit of such a task n_y we broadcast the resource h_{n_x} to all its unscheduled ascendants. The reason for choosing h_{n_x} is that its father n_x (extracted in line 4 of listing 5.1) belongs to the DS, and it is the zeroing of the edge $e_{x,y}$ that leads to a reduction of the makespan. This procedure visits all the ascendants of n_y and stops when it finds a task n_k with an already assigned resource. It sets all the DS_{desc} values of the tasks n_k to the value $tlevel(n_y, h_{n_y}) + blevel(n_y, h_{n_y})$ that is the weight of the longest path passing through n_y.

The suggested resource is not mandatory, therefore it is possible that one task might not be assigned to the suggested resource. If this happens the suggested resource is changed to the new resource assigned to that task. Note that the modification must affect all the chain of nodes visited during the suggestion process. This step can easily be performed in $O(1)$ if all the nodes receiving a suggested resource use pointers to the task that originally broadcasted the resource itself.

5.4.4 Analysis of the algorithm

Now we study the performance of the algorithm on the two primitive DAG structures: *fork* and *join*. Since any DAG can be decomposed into a collection of forks/joins and optimal schedule lengths are known for these primitives [54][80], this analysis is usually done in order to examine the approximate optimality of the algorithm. The known algorithms for computing optimal fork/join schedules work on homogeneous systems. Therefore, we restrict the analysis to the case of a homogeneous system environment. In this context there is no need to specify a particular machine, therefore we simplify the notation using τ_i in order to identify the computation time of task n_i.

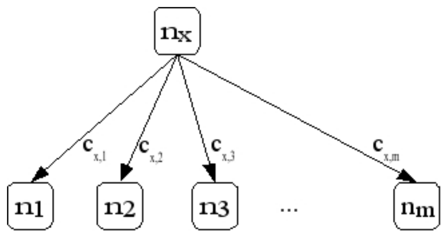


Figure 5.3: Fork DAG primitive.

Behavior on fork DAGs

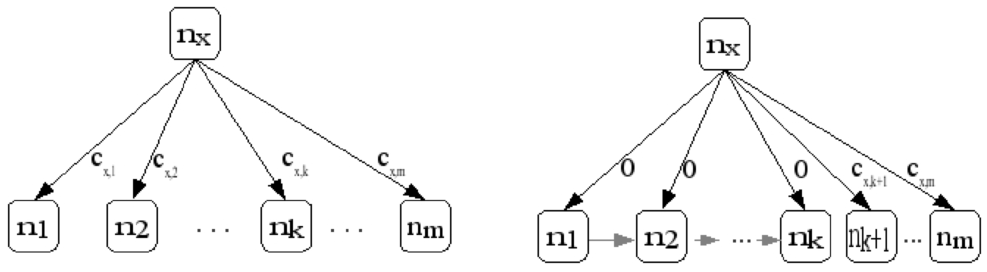
Let n_x be the root node, n_j $(j = 1...m)$ the children of n_x and $c_{x,j}$ the communication cost between n_x and n_j (figure 5.3). Assume that nodes and edges are sorted such that:

$$\tau_j + c_{x,j} \geq \tau_{j+1} + c_{x,j+1}, \qquad j = 1..m. \tag{5.3}$$

The optimum clustering [54] is given by the schedule that assigns the same resource to the first k children, where k must satisfy the following inequalities:

$$\sum_{j=1}^{k} \tau_j \leq \tau_k + c_{x,k}, \quad \sum_{j=1}^{k+1} \tau_j \geq \tau_{k+1} + c_{x,k+1}. \tag{5.4}$$

This can be easily explained like that. Call the optimal parallel time of the fork primitive PT_{opt}. Let h for $h = 1..m$ be the greatest h for which $\tau_x + \tau_h + c_{x,h} > PT_{opt}$, then $c_{x,i}$ must have been zeroed for $i = 1..h$, otherwise we have a contradiction. Consider that all other edges $i > h$ need not to be zeroed because in doing so they



(a) Initial mapping, at the beginning of the CCF algorithm.

(b) Step $k + 1$ of the CCF algorithm.

Figure 5.4: Clustering of the fork DAG primitive produced by the CCF algorithm (homogeneous systems).

do not decrease the current PT, instead they could increase it. Assume that $\tau_{m+1} = c_{x,m+1} = 0$, then the optimal PT is: $PT_{opt} = \tau_x + \max\left\{ \sum_{j=1}^{h} \tau_j, c_{x,h+1} + \tau_{h+1} \right\}$.

Now we intuitively discuss the behavior of the algorithm on a fork primitive, then we will give a formal characterization. At step 1 each node is in a unit cluster (figure 5.4(a)), task n_x is mapped on host h_x and child n_j is assigned to the host h_j by the initial mapping. n_x is the only task in the *RunningQueue*, it is extracted and its children are inserted into the *ChildQueue*. Each child is then extracted for mapping, according to the order given by the priority definition (5.2) that corresponds to the ordering defined in (5.3). For each child n_j the resource of n_x is selected if $tlevel(n_j, h_x) + blevel(n_j, h_x)$ is reduced in a maximum degree. At step k (figure 5.4(b)) the *tlevel* of the child task n_k, for which the resource of n_x is chosen, is given by:

$$tlevel(n_k, h_x) = \tau_x + \sum_{j=1}^{k-1} \tau_j.$$

Follows that the first k children of n_x are sequentialized in host h_x if $\sum_{j=1}^{k-1} \tau_j \leq c_{x,k}$.

Theorem 5.4.1. *The CCF algorithm computes optimal schedules for fork DAGs in homogeneous systems.*

Proof. Task n_x is selected from the *RunninQueue* and all its children are inserted into the *ChildQueue*. Tasks are extracted from the *ChildQueue* in descending order of their priorities, therefore they are ordered such that: $\tau_j + c_{x,j} \geq \tau_{j+1} + c_{x,j+1}$, $j = 1..(m-1)$.

For the fork primitive all the children of n_x are ready and the CCF algorithm performs a sequence of edge zeroing steps from left to right up to the point k such that: $\sum_{j=1}^{k-1} \leq c_{x,k}$ and $\sum_{j=1}^{k} \tau_j > c_{x,k+1}$. The proof that $PT_{opt} = PT_{CCF}$ is done by contradiction. Suppose that $k \neq h$ (h is the optimal edge zeroing stop point and k is the optimal edge zeroing stop point determined by CCF) and $PT_{opt} < PT_{CCF}$, then we can distinguish two cases:

1. If $h < k$ then $\sum_{j=1}^{h} \tau_j < \sum_{j=1}^{k} \tau_j \leq c_{x,k} + \tau_k \leq c_{x,h+1} + \tau_{h+1}$. But $PT_{opt} = \tau_x + c_{x,h+1} + \tau_{h+1} \geq \tau_x + c_{x,k} + \tau_k \geq \tau_x + \max\left\{ \sum_{j=1}^{k} \tau_j, c_{x,k+1} + \tau_{k+1} \right\} = PT_{CCF}$, which is a contradiction.

2. If $h > k$ then $\sum_{j=1}^{h} \tau_j \geq \sum_{j=1}^{k+1} \tau_j > c_{x,k+1} + \tau_{k+1} \geq c_{x,h+1} + \tau_{h+1}$. But $PT_{opt} = \tau_x + \sum_{j=1}^{h} \tau_j \geq \tau_x + \max\left\{ \sum_{j=1}^{k} \tau_j, c_{x,k+1} + \tau_{k+1} \right\} = PT_{CCF}$, which is a contradiction.

Therefore $PT_{opt} = PT_{CCF}$. □

Behavior on join DAGs

The algorithm assumes DAGs starting with one task, therefore we consider a *join* primitive as described by figure 5.5. A zero computation task is added, which connects with zero cost communication links the m tasks of the *join* primitive.

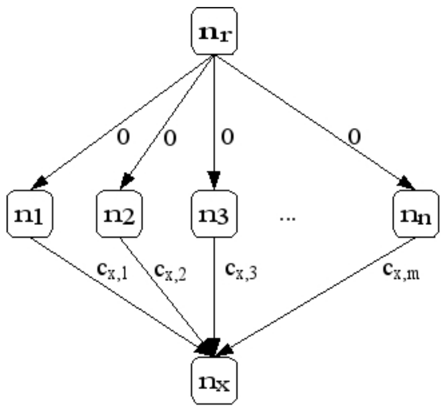


Figure 5.5: Join DAG primitive.

Remember that this analysis considers a homogeneous environment. Now we show that this first version of the proposed CCF algorithm does not produce optimal schedules for the join DAG primitive.

At step 1 node n_r is extracted from the *RunningQueue* and its children are inserted into the *ChildQueue*. Then n_1 is extracted and left in its own resource (the one assigned by the initial mapping) because there is no improvement in moving it to the resource of n_r, i.e its completion time does not change. The same happens with all the other children. Therefore, each child n_j is assigned to a unit cluster and the task n_x is mapped to the host assigned to the parent laying on the CP. Follows that the proposed algorithm is not optimal for *join* DAGs. In order to derive an optimal schedule the algorithm should implement a kind of *look-ahead* procedure, which checks for zeroing not only incoming edges of a task but also outgoing edges. In section 5.4.5 we will describe the final version of the algorithm which achieves optimality also for this primitive.

Identification of DS nodes

The heuristic implemented in the CCF algorithm is based on the identification of the DS nodes. So, we must be sure that DS nodes are correctly identified. The purpose of this subsection is to analyze the node selection process of the proposed algorithm.

Lemma 5.4.1. *Assume that $n_x = head(RunningQueue)$ at the beginning of step i, then one DS must go through one descendant of n_x.*

Proof. At the beginning of step i the parallel time is $PT_i = tlevel(n_{ds}, h_{n_{ds}}) + blevel(n_{ds}, h_{n_{ds}})$, where n_{ds} is a DS node. Let RNC be the set of tasks already extracted from the *RunningQueue* but with at least one child not mapped to a resource (because it is not ready yet) and let be $Prio\_RNC = \max_{n_j \in RNC} \{Prio\_RQ(n_j)\}$

We can distinguish two cases, $RNC = \emptyset$ and $RNC \neq \emptyset$:

1. If $RNC = \emptyset$ then $Prio\_RQ(n_x) = tlevel(n_x, h_{n_x}) + blevel(n_x, h_{n_x})$ is the maximum on all the nodes in the *RunningQueue* and n_x is a DS node itself. Follows that at least one child of n_x must lay in one DS.

2. If $RNC \neq \emptyset$ then $Prio\_RQ(n_x) \leq Prio\_RNC$. Now we have two cases:

   (a) If $Prio\_RQ(n_x) = Prio\_RNC$ then either n_x is a DS node itself (and there is more than one DS) or $Prio\_RQ(n_x)$ is given by the value DS_{desc}, set by the heaviest child n_h (which is not ready) of a task in the set RNC, through the *suggestResource*() procedure. n_h is also a descendant of n_x.

   (b) If $Prio\_RQ(n_x) < Prio\_RNC$ then if $Prio\_RQ(n_x) = tlevel(n_x, h_{n_x}) + blevel(n_x, h_{n_x})$ n_x is a DS node itself. Otherwise, if $Prio\_RQ(n_x) = DS_{desc}(n_x)$ then the DS passes through a not ready child n_k of a task in the set RNC, but, since $DS_{desc}(n_x)$ was set by the *suggestResource*() procedure starting from n_k, n_k is a descendant of n_x.

□

Lemma 5.4.1 states that the first selection phase considers nodes belonging to the DS or nodes having, as descendant, a partially free DS node. In the latter case, the purpose is to make a partially free DS node ready as soon as possible.

Monotone reduction of the schedule length

The node n_i extracted from the *RunningQueue* has the maximum value of $tlevel + blevel$. The mapping phase assigns resources to all the ready children of n_i, each time zeroing one or more incoming edges, provided that the schedule length does not increase. If one step of the algorithm is identified with both the extraction of a DS node from the *RunningQueue* and the mapping of all the ready children, then the property of monotone reduction of the schedule length holds, i.e. for each step i of the algorithm $PT_{i-1} \geq PT_i$, where PT_i is the parallel time at step i (weight of the DS at a given step i).

Theorem 5.4.2. *For each step i of the CCF algorithm $PT_i \geq PT_{i+1}$*

Proof. The parallel time at step i can be computed as: $PT_i = \max_{n_j \in V} \left\{tlevel(n_j, h_{n_j}) + blevel(n_j, h_{n_j})\right\}$. At step i a node n_x is extracted from the *RunningQueue*. All children of n_x are assigned to a temporary resource and now the algorithm tries to assign a definitive resource to all the ready children n_r. The temporary resource h_t of a child is going to change to a new resource h_n if:

$$tlevel(n_r, h_n) + blevel(n_r, h_n) < tlevel(n_r, h_t) + blevel(n_r, h_t).$$

Given this, since $PT = \max_{n_j \in V} \left\{tlevel(n_j, h_{n_j}) + blevel(n_j, h_{n_j})\right\}$ and PT_i and PT_{i+1} are computed, respectively, before and after the definitive mapping of the ready children, follows that: $PT_i \geq PT_{i+1}$. □

5.4.5 Final version of the algorithm

The final version of the algorithm aims at reaching the optimality also for the *join* DAG structure. As we will see in the experimental results section this modification, combined with the *suggestResource*() process (section 5.4.3), leads to an improvement of the quality of the scheduling produced, i.e., to a reduction of the *makespan*. To achieve this goal the final version of the algorithm implements, in the *assignResource()* function, a *look-ahead* strategy, similar to the one used in the DCP algorithm [78]. In particular, we first try to select a resource basing the decision on the minimization of the cost function $tlevel + blevel$ of the task n_j (described in section 5.4.3) then, if the schedule length is not reduced, the *assignResource()* procedure is repeated, but this time trying to virtually schedule the heaviest child n_{hc} of n_j (the one that determines the *blevel* of n_j) in the same resource. Finally, a *suggestResource*() is called from n_{hc} suggesting the resource selected for n_j to all the unmapped ascendants. The look-ahead strategy defines the cost function as follows:

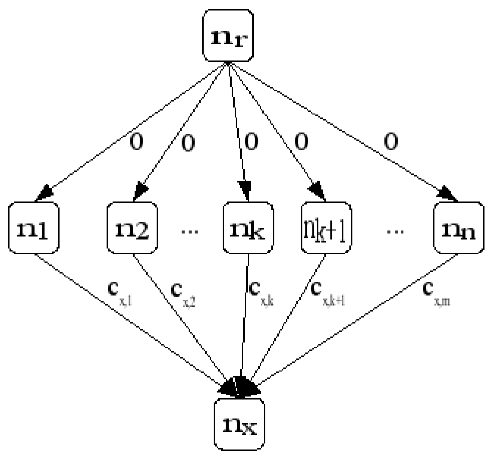
$$\begin{aligned}Cost(n_j, h) &= tlevel(n_j, h) + blevel(n_j, h)\\ &\quad + tlevel(n_{hc}, h) + blevel(n_{hc}, h)\end{aligned}$$

where n_{hc} is the heaviest child of n_j and h the resource that is under examination. In this way, the selected resource for n_j tries to maximize the probability of zeroing multiple edges for its heaviest child. With this look-ahead strategy, combined with the resource suggestion process, the algorithm achieves the optimality for the DAG join primitive and experimental tests show improvements on the length of the schedule produced. On the other side, the monotone reduction property does not hold anymore. This happens because the resource is selected considering the estimate reduction of the schedule length, given by virtually mapping the heaviest child on the same resource. The mapped task is n_j (not n_{hc}) and the variation in $tlevel(n_j, h) + blevel(n_j, h)$ is the one that concretely affects the schedule length. Therefore, when mapping n_j it can happen that the total schedule length increases. Since the future mapping of the heaviest child does not guarantee a reduction greater than the increment possibly obtained before, the monotone reduction property does not hold anymore. We have experimentally found that if the look-ahead strategy is applied systematically each time a resource is selected (like in the DCP algorithm) then there is no improvement by using this technique. Otherwise, if it is used only on a restricted class of events then good results can be obtained. The proposed algorithm uses the look-ahead strategy only in the case in which the schedule length is not reduced by the mapping function, and experimental results show a good improvement in the quality of the schedule produced.

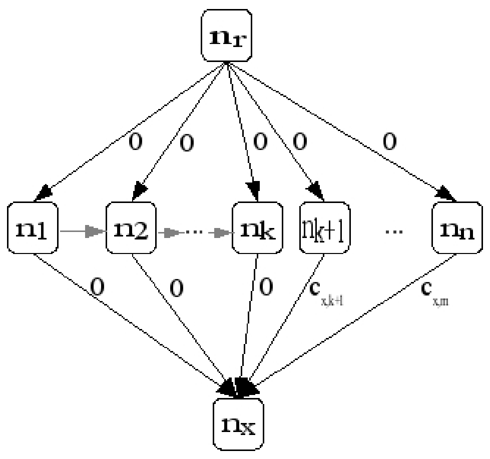
The optimality for the *fork* DAG primitive still holds for this final version and the next paragraph shows how the optimality for the *join* primitive is obtained.

The *suggestResource()* function

In the final version of the algorithm, with the introduction of the look-ahead process, the suggested resource plays a central role. Now we can see why its effect is to



(a) Initial mapping, at the beginning of the CCF algorithm.



(b) Step $k + 1$ of the CCF algorithm.

Figure 5.6: Clustering of the join DAG primitive produced by the CCF algorithm (homogeneous systems).

increase the probability of zeroing multiple incoming edges. The look-ahead strategy tries to map a task to a resource by looking forward, i.e. considering the outgoing edge of the task belonging to the DS. In this way, the value of the cost function can be considerably reduced by the "virtual" zeroing of edges connecting the DS child to parents mapped in the same resource. For example, consider figure 5.2. Task n_6 is the first one considered for mapping, but it is not ready, thus it waits and suggests the resource h_{n_1} to all its ascendants. When tasks n_2, n_4 and n_5 are considered for mapping, the algorithm first tries to schedule them on their parents resources (giving priority to the zeroing of the incoming edge) but if it is not successful in doing this, it tries with the look-ahead strategy, this time giving priority to the zeroing of the outgoing edge. For task n_2 there is no improvement in using the suggested resource because it is duplicated, since the reference task is n_1, which is the parent of n_2. For tasks n_4 and n_5, instead, the suggested resource may represent a real alternative. If the communication is heavy the improvement in moving these tasks, together with the child n_6, to the resource h_{n_1} could be high. In fact, edges $e_{1,6}$, $e_{4,6}$, $e_{5,6}$ and eventually $e_{2,6}$ (if n_2 is mapped to h_{n_1}) can be zeroed.

5.4.6 Join DAG analysis for the final version of the algorithm

The algorithm assumes DAG with one entry node and figure 5.5 shows the *join* primitive considered for the analysis: a root node n_r with zero computation time connects all the entry nodes of the *join* primitive with zero cost transfer times. Assume that nodes and edges are sorted such that:

$$\tau_j + c_{j,x} \geq \tau_{j+1} + c_{j+1,x} \qquad j = 1..m-1. \tag{5.5}$$

As for the *fork* primitive, the optimum clustering [80] is a tradeoff between parallelization and sequentialization; it is given by the schedule that clusters together the first k nodes n_j, where k must satisfy the following inequalities:

$$\sum_{j=1}^{k} \tau_j \leq \tau_k + c_{k,x}, \quad \sum_{j=1}^{k+1} \tau_j \geq \tau_{k+1} + c_{k+1,x}.$$

The first node in the *RunningQueue*, n_r, is extracted and all its children n_j ($j = 1..m$) are inserted into the *ChildQueue*. Since all the communication costs $c_{r,j}$ are zero, tasks n_j are extracted from the *ChildQueue* following the order given by their *blevel* that corresponds to the order defined in (5.5). Task n_1 is extracted and, since there is no improvement in assigning it to h_r, the look-ahead strategy is called. The resource assigned to n_1 is still h_{n_1} but this time the *suggestResource*() suggests resource h_{n_1} to all the n_j. The look-ahead strategy is called for all the remaining tasks and $c_{j,x}$ is checked for zeroing by virtually mapping n_x to the suggested resource. The first k nodes are sequentialized on the suggested resource if $\sum_{j=1}^{k-1} \tau_{j,h_{n_1}} \leq c_{k,x}$. Finally, task n_x is assigned to h_{n_1} since its first k incoming edges can be zeroed.

Theorem 5.4.3. *The final version of the CCF algorithm computes optimal schedules for join DAGs in homogeneous systems.*

Proof. Task n_r is selected from the *RunninQueue* and all its children are inserted into the *ChildQueue*. Tasks are extracted from the *ChildQueue* in descending order of their priorities, therefore they are ordered such that: $\tau_j + c_{j,x} \geq \tau_{j+1} + c_{j+1,x}$, $j = 1..(m-1)$. The *ChildQueue* orders the tasks on the sum $tlevel + blevel$. Since the incoming edges of the tasks n_j have a label value of zero, the *blevel*, and hence the outgoing edge weight, is the one that affects the ordering.

The first task n_1 is extracted from the *ChildQueue* and the *assignResource*() function tries to schedule it on the resource of its father. There is no improvement in moving n_1, therefore the look-ahead procedure is called. Also this time the resource of the father n_r and the resource h_{n_1}, assigned to n_1 by the initial mapping, give the same value for the cost function and task n_1 is definitively mapped to h_{n_1}. The *suggestResource*() function is called and the resource h_{n_1} is suggested to all the other tasks n_j, $j = 1...n$. Now task n_2 is extracted from the *ChildQueue* and, like was happened for n_1, it tested of its father resource with no improvements and the look-ahead strategy is called. The algorithm finds that the cost function is minimized on the suggested resource h_{n_1}, because n_x is virtually mapped to h_{n_1} and two edges can be zeroed. This process goes on by mapping tasks n_j to the resource h_{n_1} while performing a sequence of (virtual) edge zeroing steps from left to right up to the point k such that: $\sum_{j=1}^{k-1} \leq c_{k,x}$ and $\sum_{j=1}^{k} \tau_j > c_{k+1,x}$.

The proof that $PT_{opt} = PT_{CCF}$ is done by contradiction. Suppose that $k \neq h$ (h is the optimal edge zeroing stop point and k is the optimal edge zeroing stop point determined by CCF) and $PT_{opt} < PT_{CCF}$, then we can distinguish two cases:

1. If $h < k$ then $\sum_{j=1}^{h} \tau_j < \sum_{j=1}^{k} \tau_j \leq c_{k,x} + \tau_k \leq c_{h+1,x} + \tau_{h+1}$. But $PT_{opt} = \tau_x + c_{h+1,x} + \tau_{h+1} \geq \tau_x + c_{k,x} + \tau_k \geq \tau_x + \max\left\{ \sum_{j=1}^{k} \tau_j, c_{k+1,x} + \tau_{k+1} \right\} = PT_{CCF}$, which is a contradiction.

2. If $h > k$ then $\sum_{j=1}^{h} \tau_j \geq \sum_{j=1}^{k+1} \tau_j > c_{k+1,x} + \tau_{k+1} \geq c_{h+1,x} + \tau_{h+1}$. But $PT_{opt} = \tau_x + \sum_{j=1}^{h} \tau_j \geq \tau_x + \max\left\{ \sum_{j=1}^{k} \tau_j, c_{k+1,x} + \tau_{k+1} \right\} = PT_{CCF}$, which is a contradiction.

Therefore $PT_{opt} = PT_{CCF}$. □

5.5 A variant of the DSC algorithm

In this section we propose a variant of the DSC (Dominant Sequence Clustering) algorithm. First of all we briefly recall how the original algorithm works. The DSC [135] selects at each step a node belonging to the DS, the critical path of the scheduled DAG computed as the maximum value of *tlevel* + *blevel*, and assigns it to the resource that allows the minimum start time. In this first version the algorithm is simple and it achieves good results but it is not optimal for join DAGs. The authors formulate a final version adding two features: the possibility to cluster together already scheduled parents of a node and the possibility to book a resource for a partially free node (a node for which only some of its parents are visited) belonging to the DS. The final version of the algorithm achieves optimality also for join DAGs (in homogeneous systems). We have adapted the DSC algorithms in order to work in a heterogeneous environment and we have tested the two versions of the algorithm with random DAGs and different network topologies. The results showed no real improvements from the first and the final version of the algorithm, on average the first version performs better than the final one.

Here we propose a variant of the DSC algorithm that leverages from some concepts used in the final version of the CCF algorithm. In particular, we take the first version of the DSC algorithm and we add the look-ahead strategy together with the *suggestResource*() function. These two features are integrated as follows:

1. Like for the CCF algorithm the look-ahead process is called only when the algorithm does not find a better resource than the one assigned by the initial mapping.

2. The *suggestResource*() function is used only when a resource is selected with the look-ahead process. In particular, the child task that is virtually mapped to the resource just assigned to the father task, is the one that calls this function.

The result is a variant of the DSC algorithm for which some of the properties of the original DSC algorithm still holds, like optimal behavior on fork and join DAGs in homogeneous systems and the correctness in locating DS nodes. Due to the look-ahead strategy, like for the CCF algorithm, the monotone reduction property of the schedule length does not hold anymore and the time complexity of this new variant of the DSC algorithm is no longer $O((v + e) \log v)$.

As can be seen in the next chapter experimental results show significant improvements on the length of the schedules produced, i.e. reduction of the makespan.

5.6 Summary

In this chapter we have presented two new DAG scheduling algorithms designed to work in a Grid computing system, which is a heterogeneous computing system with a heterogeneous interconnection network.

The first proposed algorithm is called CCF (Cluster ready Children First)and it is a list scheduling algorithm. It visits the DAG in topological order and uses a two step node selection phase based on a priority that identifies tasks belonging to the Dominant Sequence, if it is possible, otherwise means that the next DS task is blocked because it is not ready and in this case it is selected the task (or a sequence of tasks) allowing the DS node to become ready as soon as possible. More specifically, the first step selects a running task (i.e., already mapped) and then all the ready children of that task are selected for mapping. This selection of tasks is different from all the other list scheduling approaches. The first selection step considers already mapped tasks and in the second step can happen that only non-DS tasks are mapped. This makes this heuristic a hybrid between CP-based and non CP-based. Then, the final version of the algorithm, introduces a process for adding a particular resource to the set of candidate resources (the set from which the best resource for a task is drawn) and a look-ahead strategy is invoked when the algorithm does not find a better resource other than the initial mapping.

Some properties of the proposed CCF algorithm are identified and analyzed, such as the optimal behavior on fork and join DAGs in homogeneous systems, monotone reduction of the schedule length (first version of the algorithm) and the correct identification of DS nodes.

The second proposed algorithm, the DSC_VAR, is a variant of the famous DSC. DSC_VAR adds to the first version of the original algorithm the resource suggestion process and the look-ahead strategy used in the CCF algorithm. Some of the properties of the original version still hold for this variant, like the optimal behavior on fork and join DAGs in homogeneous systems.

The next chapter describes experimental results conducted with these two algorithms. Random DAGs and different network topologies are used in order to compare the results of the proposed algorithms with other two algorithms: the original DSC and the Hybrid Remapper. The platforms describing the heterogeneous computing environment used in the simulations are created using the GridG toolkit. As we will see, the performance comparison shows improvements on the length of the schedules produced by the two proposed algorithms, both for poor and good initial mappings.

6 Experimental Results

Scheduling the tasks of a distributed application has been an active field of research for several decades. The classic scheduling problem is to find an assignment of application tasks onto a set of distributed resources with the aim of minimizing the makespan, i.e. the overall completion time of the parallel application. As we have seen in previous chapters, it has been shown that non-trivial instances of the DAG scheduling problem are NP-complete. As a result, the researchers have proposed different approaches to solve the problem. However it is difficult to determine which solutions are practical for real scenarios. Given the NP-completeness nature of the problem, it is really hard to make a theoretical evaluation of the different proposed heuristics. On the other hand, studying the behavior of scheduling algorithms on real platforms, although providing accurate and undoubtedly realistic results, is not practical due the dynamic availability of the resources. The main problem is that experiments on real platforms are often non-reproducible, this means that results of different scheduling algorithms are not comparable. Therefore, simulation is the best choice in order to test and compare scheduling algorithms.

This chapter presents experimental results obtained with the simulation of the proposed algorithms CCF and DSC_VAR, which are compared to the original DSC [135] and the Hybrid Remapper [90] algorithms.

6.1 Simulation framework

We have conducted extensive simulations to evaluate the performance of the two proposed scheduling algorithms. In particular, we present a comparison of the performance results of the CCF and DSC_VAR algorithms with other two scheduling algorithms, the original DSC [135] described in section 4.6.9 and the Hybrid Remapper [90] described in section 4.6.14. The Hybrid Remapper is presented by its authors in three versions called PS, CS and CD. Here we use the third version (CD), which is the one that considers dynamic values in the computation of priorities.

The input to the simulator consists of a set of DAGs and an environment description of the network and the resources (the platform). Each scheduling algorithm can retrieve informations like: machine characteristics, latency and bandwidth. In the following the experimental session will be described considering three main areas: the simulator, DAG generation and platform generation. Finally results and comments of the performance comparison are presented.

6.1.1 The simulator

We evaluated three simulators: GridSim [15], Simgrid [17, 81, 65] and MetaSimgrid [82]. With Simgrid it is possible to model a platform and defining arbitrary topology for the interconnection network, whereas with GridSim this cannot be done. On the other hand with Simgrid the platform model must be constructed by hand; it can be done for few hosts, but for more realistic cases it is practically impossible. Furthermore, Simgrid lacks a number of convenient features to craft simulations of a distributed application where scheduling decisions are not taken by a single process. MetaSimgrid is a Grid simulation tool build on top of Simgrid that overcomes to some limitations of Simgrid. For example, it is possible to import an existing platform. In this way, it is possible to use existing tools for building platforms with *approximately* realistic topology, including switches and routers, and to gather data on its available resources. Examples of such a tools are:

- ENV with NWS: ENV (Effective Network View) [113] allows to discover the effective topology of a network from a given host, whereas NWS (Network Weather System) [130] can be used to gather all the traces (link latency, bandwidth, cPU load, etc.) needed to simulate external load.
- GridG with Tier: GridG [88] is a toolkit based on Tier [16, 33]. With this toolkit it is possible to easily generate *approximately* realistic plaforms with arbitrary network topology and resources characteristics.

Our final choice was to use MetaSimgrid for the simulation of the algorithms and the GridG toolkit for the generation of the test platforms.

6.1.2 DAG generation

Due to the NP-completeness of this scheduling problem, heuristic ideas used in CCF and DSC_VAR cannot always lead to an optimal solution. Thus it is necessary to compare the average performance of different algorithms using randomly generated graphs.

We have generated each DAG at random, starting from he following input parameters: number of nodes, number of edges, range (minimum and maximum) of computation values and range of data transfer values. These values are abstract labels (number of operations and size of data). We are dealing with DAGs, therefore the number of edges $|E|$ is limited by:

$$|E| \leq |V|(|V|-1)/2$$

where $|V|$ is the number of nodes. Random edges (i, j) are generated with $i < j$. Nodes and edges labels are generated at random and their values are drawn from the corresponding ranges given as input. Abstract labels are transformed in time values by the scheduling algorithm once the characteristics of the resources are known. In particular, in order to obtain the estimate of the execution time of a task, we have to retrieve the CPU speed (given in mips) and to divide the abstract label associated to that task by the CPU speed. Analogously, in order to obtain the estimate of the transfer time we have to retrieve the link characteristics (latency and bandwidth) and to perform this simple operation:

$$Transfer\_time\_estimate = latency + (data\_size/bandwidth).$$

The *Communication Computation Ratio* (CCR) is defined as the average edge weight divided by the average node weight. DAGs with a CCR smaller than 1 are called coarse grain DAGs and DAGs with a CCR grater than 1 are called fine grain DAGs. All the DAGs used for the tests are divided into two groups: the first group contains coarse grain DAGs with a CCR ranging from 0.05 up to 1 and the second group contains fine grain DAGs with a CCR ranging from 1 up to 20. Each group is divided into 30 subgroups, each containing 20 DAGs with (approximately) the same CCR. The CCR of the 30 subgroups progressively increases from subgroup 1 to subgroup 30, i.e. from 0.05 to 1 for the coarse grain DAGs and from 1 to 20 for the fine grain DAGs.

6.1.3 Platform generation

A realistic platform is essential in evaluating middleware for computational grids. The platform is the raw Grid, an annotated graph representing the network topology and the hardware and software available on each node and link within it. For the generation of the network topology and the definition of the characteristics of the resources we have used the GridG toolkit [88]. It leverages from Tier [16, 33], a tool for network topology generation, extending it in order to produce graphs that conform to recently discovered power laws of Internet topology [37]. In general, a Grid generator has to produce a grid of a given number of hosts and must meet the following requirements:

- *Realistic network topology*: real network topologies are connected and have hierarchical structures. Furthermore recent studies found that wide area network topologies follow certain topological power laws.

- *Realistic annotations for hosts and network components*: that means providing fundamental characteristics for hosts, links, routers and switches. For hosts such characteristics are: architecture type, processor type and speed, number of processors, memory size, disk size, hardware vendor, operating system, and available software. For a link, it should provide the hardware bandwidth and latency. For routers and switches, it should specify the aggregate backplane or switching fabric throughput and latency. Furthermore, characteristics of different components are often correlated, e.g. memory size increases with processor speed, a high speed router is unlikely to be connected only to a few slow links, and so on.

Power Laws	Rank exponent	$d_v \propto r_v^R$
	Outdegree exponent	$f_d \propto d^O$
	Eigen exponent	$\lambda_i \propto i^\epsilon$
Approximation	Hop-plot exponent	$P(h) \propto h^H$

Table 6.1: Power laws of Internet topology [88, 37].

GridG starts with the output of a structure-oriented topology generator (they currently use Tiers [16, 33]) and adds redundancy to it in such a way as to make it conform to the power laws. Table 6.1 summarizes the three power laws identified by Faloutsos et al. in [37]. Faloutsos defines the rank of a node in the following way. First the nodes are sorted in decreasing order of out-degree, then ranks are assigned according to this ordering and the number of nodes in each equivalence class (classes group nodes with the same out-degree). Nodes belonging to the highest out-degree class have rank $r_v = 1$, nodes belonging to the second highest out-degree class have rank $r_v = 1 + (number\ of\ nodes\ in\ the\ previous\ class)$.

The rank exponential law says that the out-degree d_v of a node v, is proportional to the rank of the node (r_v) raised to the power of a constant R.

The out-degree exponent law says that the frequency f_d of an out-degree d is proportional to the out-degree raised to the power of a constant O.

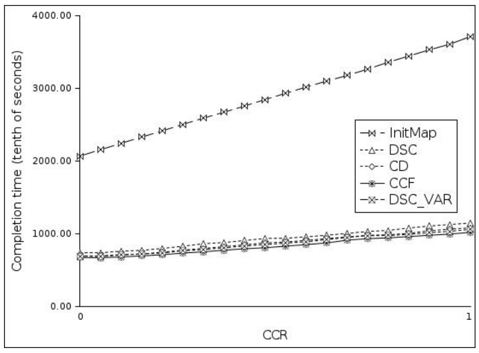
The Eigen exponent power law says that the eigenvalues λ_i of a graph are proportional to the order i raised to the power of a constant ϵ.

The hop-plot exponent power law is considered as an approximation and says that the total number of pairs of nodes $P(h)$, within h hops, is proportional to the number of hops raised to the power of a constant H.

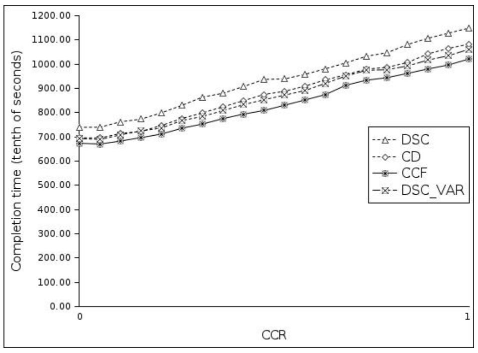
With the GridG toolkit we have created different platforms with the number of resources ranging from 90 to 614. In order to consider different conditions of environment load, for each platform we have generated 10 variants with weights (cpu and bandwidth) reduced at random by a percentage that ranges from 10% (for the first variant) to 95% (for the last variant). Final execution times for a platform are an average of the execution times on the variants.

6.2 Performance comparisons

As mentioned above the performance comparison of the proposed algorithms is made using two input sets: the first containing coarse grain DAGs (CCR smaller than 1) and the second containing fine grain DAGs (CCR greater than 1). Tests were performed over different network topologies; here we present results for a platform formed by 160 resources. For each single simulation the execution times are computed as an average of the execution of the 20 DAGs of each subgroup, having different number of nodes and edges but with, approximately, the same CCR. In order to consider different loads of the environment, each simulation is repeated 10 times with different



(a) Results for InitMap, DSC, CD, DSC_VAR and CCF.



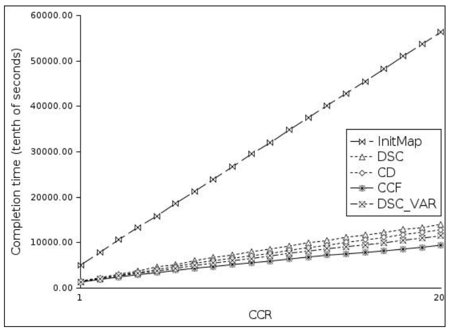
(b) Results for DSC, CD, DSC_VAR and CCF.

Figure 6.1: Results for CCR ranging from 0.05 to 1.

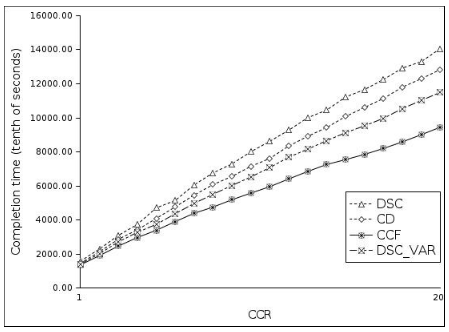
values of loads for the machines and the network, and the final execution time, for a given CCR, is the average of the 10 simulations.

All the algorithms considered run over an initial mapping, which assigns a resource to each task. We call the algorithm that implements this initial step the InitMap. All the considered scheduling algorithms try to improve this initial schedule. We use an implementation of the InitMap that visits the DAG in topological order and assigns to each task the resource that allows the earliest start time for that task. In order to study how the proposed heuristics can improve an initial bad mapping we have added the constraint of using a resource only once. At the end of this section we will consider how the scheduling algorithms behave on an improved version of the InitMap.

Figure 6.1 shows the results for an increasing CCR ranging from 0.05 to 1. In particular, figure 6.1a considers all the tested algorithms, whereas figure 6.1b excludes

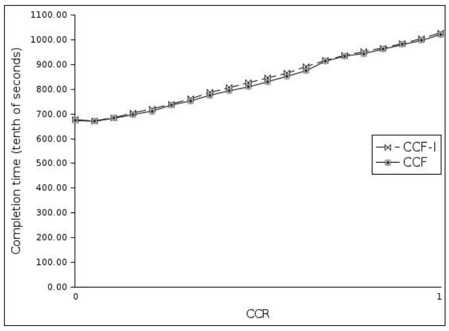


(a) Results for Initial mapping, ETF, DSC, CD, DSC_VAR and CCF.

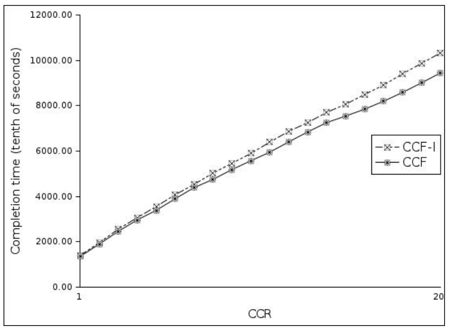


(b) Results for DSC, CD, DSC_VAR and CCF.

Figure 6.2: Results for CCR ranging from 1 to 20.



(a) Coarse grain DAGs (CCR smaller than 1).



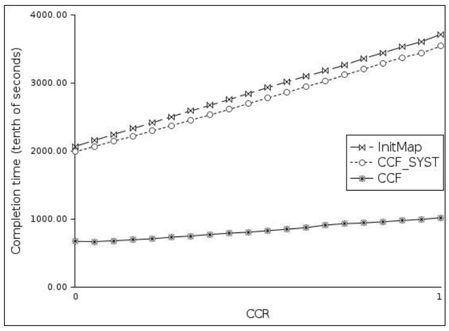
(b) Fine grain DAGs, (CCR greater than 1).

Figure 6.3: Comparison between the first version and the final version of the CCF algorithm.

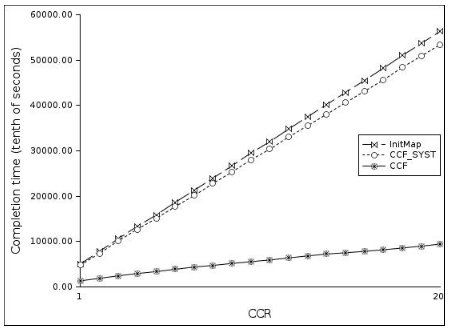
the InitMap algorithm in order to focus on the results of the other algorithms. The results for DSC, CD, DSC_VAR and CCF are close to each other, but the proposed CCF and DSC_VAR outperform, on average, all the other algorithms, as can be seen from figure 6.1b. For coarse grain DAGs the difference is small, in particular the DSC_VAR is very close to the CD algorithm. The original version of the DSC algorithm perform worse than proposed variant DSC_VAR.

Figure 6.2 considers simulation results for fine grain DAGs, where the average communication time is greater than the average computation time. Figure 6.2a shows the results of all the tested algorithms and figure 6.2b focuses on the DSC, CD, DSC_VAR and CCF algorithms. Also for this class of DAGs the DSC_VAR and CCF algorithms outperform, on average, all the other algorithms. In particular the improvement gained by these two algorithms is higher than in the case of coarse grain DAGs. This result is mainly due to the application of the looking-ahead strategy, which optimizes the selection of a resource considering the amount of communication between a task and its heaviest child, together with the suggestion resource process, which tracks the resource that allows to increase the probability of zeroing multiple incoming edges of the heaviest child. As can be seen, the improvement gained increases with the CCR. The DSC_VAR still performs better of the original version of DSC.

Figure 6.3 shows the performance results of the first and final version of the CCF algorithm. Remember that the final version differs from the first version because it implements the looking-ahead strategy, with the aim to reduce the makespan by reducing communication times. For coarse grain DAGs (figure 6.3(a)), where the average computation is higher than the average communication, the two versions perform almost the same but, as the CCR increases, and hence the communication gets higher, the final version of the algorithm outperforms the first version (figure 6.3(b)). The looking-ahead strategy is applied only when the normal selection phase do not find a better resource than the one chosen by the initial mapping. This point is crucial. As we noted in section 5.4.5, the DCP algorithm [78] implements a similar



(a) Coarse grain DAGs, i.e. CCR smaller than 1.

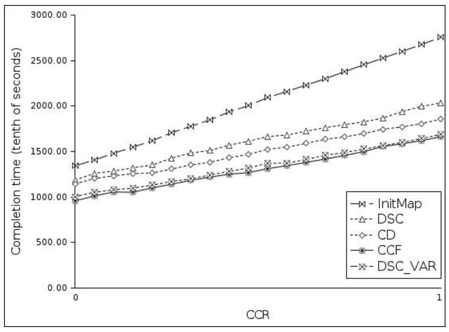


(b) Fine grain DAGs, CCR greater than 1.

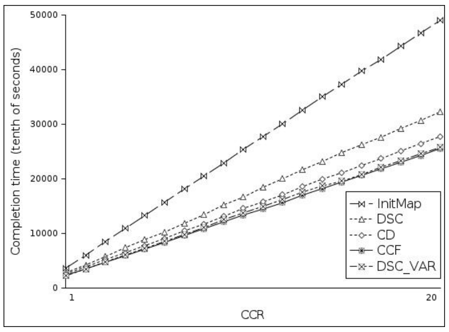
Figure 6.4: Comparison between the CCF algorithm and CCF_SYST, a version of CCF that uses the looking-ahead strategy systematically on each resource selection.

technique every time a resource is tested for mapping. When using the looking-ahead strategy, the monotone reduction property of the schedule length does not hold anymore. Therefore, another interesting analysis of the CCF algorithm, considers the comparison of the first version and a version where the looking-ahead strategy is applied systematically in each selection phase. This comparison is presented in figure 6.4. CCF_SYST is the version that applies the looking-ahead strategy systematically at each selection phase. Figure 6.4(a) considers DAGs with CCR less than one, whereas figure 6.4(b) considers DAGs with CCR greater than one. The result is that the CCF_SYST performs significantly worse than the first basic versions of CCF. In particular, it is very close to the InitMap algorithm. Remember that this version of the InitMap is the one designed to obtain poor schedules.

These results confirm that this kind of strategy must be applied with care. This



(a) Coarse grain DAGs, CCR smaller than 1.



(b) Fine grain DAGs, CCR greater than 1.

Figure 6.5: Tests with an improved initial mapping.

can be explained by the fact that in this case the monotone reduction of the schedule length does not hold anymore. In particular, tests using an improved version of the InitMap show that sometimes the schedule produced by the CCF_SYST are worse than the one obtained with the InitMap.

Finally we have tested the algorithms with a modified version of the InitMap that removes the constraint of using a resource only once and assigns to each task the resource allowing the smallest completion time (start time plus execution time). The results can be seen in figure 6.5(a) (for DAGs with CCR smaller than one) and figure 6.5(b) (for DAGs with CCR greater than one). This time the difference among the initial mapping and the other scheduling algorithms is lower, and the same is true also comparing the algorithms to each other. Anyway, the CCF and DSC_VAR still outperform the DSC and the Hybrid Remapper and their behavior is close to each other.

We also tested the proposed algorithms with other different platform definitions. The obtained results in these cases are similar. On average the DSC_VAR and CCF algorithms give better performance results in particular for fine grain DAGs. In some cases improvements are lower, and in other cases they are higher.

6.3 Summary

In this chapter we have presented the results of the experimental tests performed with the proposed algorithms CCF and DSC_VAR. We have launched the algorithms in a simulated Grid environment. In fact, given the NP-completeness nature of the problem, it is really hard to make a theoretical evaluation of the different proposed heuristics. Evaluation on real platforms is very hard, since this kind of experiments are often non-reproducible and, in order to compare the results of different algorithms, same environment conditions are needed. This is the main reason for choosing a simulated environment as testbed for the comparison of the algorithms.

We have evaluated different simulators and finally we decided to use MetaSimgrid. This simulator allows to import platforms, consisting in an annotated graph representing the network topology and the hardware and software available on each node and link within it, from external tools.

The definition of the platform is also a crucial point. In fact, in order to produce meaningful results, the platform should describe a realistic Grid. To generate the environment description we have used the GridG toolkit, that leverages from the network topology generator Tier. Basically, GridG extends Tier in order to produce graphs that conform to recently discovered power laws of Internet topology.

The input test set is formed by random generated DAGs grouped by their *Communication Computation Ratio* (CCR). Two main groups can be identified: one containing coarse grain DAGs, i.e. with CCR less than one, and one containing fine grain DAGs, i.e. with CCR greater than one. Each group is divided into 30 subgroups, each formed by DAGs with approximately the same CCR, ordered by increasing CCR. The input to the simulator consists of the set of DAGs and the platform.

We have executed different tests comparing the results of the four algorithms DSC, Hybrid Remapper, CCF and DSC_VAR considering different platforms and different initial mappings. On average, the two proposed algorithms CCF and DSC_VAR have outperformed the other two reference algorithms Hybrid Remapper and DSC. We have also proposed some tests with the aim to analyze the techniques implemented in the CCF algorithm.

Conclusions

In this thesis we have investigated the problem of scheduling parallel applications described by directed acyclic graphs (DAGs) in Grid computing systems. Grid is evolving as a service oriented architecture. In this context, the "software as a service" approach results in a componentized view of software applications and workflow can naturally be used as a component composition mechanism. Actually, the most common Grid workflow can be modelled as a DAG, where the order of execution of tasks (modelled as nodes) is determined by dependencies (in turn modelled as directed arcs). The DAG scheduling problem is showed to be NP-complete, therefore the most popular approach to compute solutions is to use heuristics. Usually these heuristics are implemented using the so called *list scheduling* technique. In list scheduling each task is assigned to a priority value and higher priority tasks are selected for mapping first. An algorithm is *static* if it computes priorities once at the beginning, otherwise it is *dynamic* and priority values are updated during the execution of the algorithm.

The main result of the thesis is the design of a DAG scheduling algorithm for Grid computing systems. The proposed algorithm is called CCF (Cluster ready children first) and it belongs to the class of dynamic list scheduling algorithms. We have first implemented and simulated some algorithms found in the literature. After an analysis of the results we have identified the algorithm showing better performance, in terms of reduction of the schedule length. These algorithms was the DSC (Dominant Sequence Clustering) and the Hybrid Remapper. Both these algorithms implements a heuristic based on the identification of nodes belonging to the DS (Dominant Sequence). The DS nodes are the most important because they define the makespan (length of the schedule) of the DAG. Although these kind of heuristics are the ones that give, on average, the best results in terms of quality of the schedule produced, they can get trapped in a locally optimal decision, leading to a non-optimal global solution. Our approach was to formulate an heuristic that is a mixture between a CP-based and a non-CP-based heuristic. In particular, the DAG is visited in topological order and we implemented a two step task selection. In the first step the algorithms selects a node belonging to the DS. What is different from all the other approaches is that the DS node is drawn from a set of already mapped nodes, i.e. a set containing frontier nodes that are visited nodes with non-visited children. The second step maps all the ready children of the selected task to a suitable resource. Children which are not ready wait for as long as all their parents are mapped. DS nodes in the first step are identified by the value of the sum of their top level plus bottom level. With this method three possible situations can happen:

- The selected node belongs to the DS and the next step maps the heaviest child.
- The selected node belongs to the DS and the next step doesn't map the heaviest child because it is not ready.

- The selected node doesn't belong to the DS because the DS node is partially free.

To address the last case where the DS node is partially free, we have implemented a technique that give higher priority to tasks that allows the partially free DS node to become ready as soon as possible. In its final version, the proposed algorithm implements a looking-ahead strategy to guide the selection of a resource considering not only the minimization of incoming data transfers of a task, but also the outgoing data transfer to its heaviest child.

Some of the techniques implemented in the CCF algorithm were reused leading to the proposal of a modified version of the DSC algorithm.

In order to evaluate the two proposed algorithms we have conducted extensive tests by simulating their execution and comparing the results with other two reference algorithms. One important point of this phase was the definition of the platform, that is the description of the networked computing system. In order to produce meaningful results, the platform should describe a realistic Grid. To generate the environment description we have used the GridG toolkit, that leverages from the network topology generator Tier. Basically, GridG extends Tier in order to produce graphs that conform to recently discovered power laws of Internet topology.

Experimental tests showed a good behavior of the two proposed algorithms CCF and DSC_VAR. On average they have outperformed the other reference algorithms Hybrid Remapper and DSC. These results were confirmed by repeating these tests under different network topologies, dimensions (number of resources) and load conditions.

Bibliography

[1] T. Adam, K.M. Chandy, and J.R. Dickson. A comparison of list schedules for parallel processing systems. *CACM*, 17(12):685–690, 1974.

[2] I. Ahmad and Y.-K. Kwok. On exploiting task duplication in parallel program scheduling. *IEEE Trans. Parallel and Distributed Systems*, 9(9):872–892, September 1998.

[3] I. Ahmad, Y. K. Kwok, M. Y. Wu, and W. Shu. Automatic parallelization and scheduling of programs on multiprocessors using CASCH. *Proc. 1997 Inter. Conf. Parallel Processing*, pages 288–291, August 1997.

[4] H.H. Ali and H. El-Rewini. The time complexity of scheduling interval orders with communication is polynomial. *Parallel Processing Letters*, 3(1):53–58, 1993.

[5] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster. The globus striped gridftp framework and server. In *Proceedings of Super Computing 2005 (SC05)*, November 2005.

[6] G. Allen, T. Dramlitsch, I. Foster, N. Karonis, M. Ripeanu, Ed. Seidel, and B. Toonen. Supporting efficient execution in heterogeneous distributed computing environments with cactus and globus. *Supercomputing 2001*, August 2001. Winning Paper for Gordon Bell Prize (Special Category).

[7] J. Almond and D. Snelling. UNICORE: Uniform access to supercomputing as an element of electronic commerce. *Future Generation Computer Systems*, 15:539–548.

[8] Sergio Andreozzi, Cristina Vistoli, and Massimo Sgaravatto. Sharing a conceptual model of Grid resources and services. In *Computing in High Energy and Nuclear Physics*, La Jolla, California, 2003.

[9] F. D. Anger, J. J. Hwang, and Y. C. Chow. Scheduling with sufficiently loosely coupled processors. *Journal of Parallel and Distributed Computing*, 9:87–92, 1990.

[10] R. Baraglia, S. Orlando, and R. Perego. Resource Management Systems: Scheduling of Resource-Intensive Multi-Component Applications. Deliverable of the Grid.it project - WP 8, January, 29 2004.

[11] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and

R. Wolski. The grads project: Software support for high-level grid application development. *International Journal of High Performance Computing Applications*, 15(4):327–344, 2001.

[12] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, S. Spring, A. Su, and D. Zagorodnov. Adaptive computing on the grid using apples. *IEEE Trans. on Parallel and Distributed Systems (TPDS)*, 14(4):369–382, 2003.

[13] F.D. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-level scheduling on distributed heterogeneous networks. In ACM Press, editor, *Proceedings of the 1996 ACM/IEEE conference on Supercomputing,*, 1996.

[14] R. Buyya, D. Abramson, and J. Giddy. An economy driven resource management architecture for global computational power grids. In *Proc. of International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000)*, 2000.

[15] R. Buyya and M. Murshed. Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *The Journal of Concurrency and Computation: Practice and Experience (CCPE)*, pages 1–32, May 2002.

[16] K. L. Calvert and M. B. Doar. Modeling internet topology. *IEEE Communications Magazine*, 1997.

[17] H. Casanova. Simgrid: A toolkit for the simulation of application scheduling. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid'01)*, pages 430–437, May 2001.

[18] T.L. Casavant and J.G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Trans. Software Engineering*, 14(2):141–154, February 1988.

[19] David Chappell and Tyler Jewell. *Java Web Services.* O'Reilly & Associates, March 2003.

[20] A.L. Chervenak, N. Palavalli, S. Bharathi, C. Kesselman, and R. Schwartzkopf. Performance and scalability of a replica location service. In *Proceedings of the International IEEE Symposium on High Performance Distributed Computing (HPDC-13)*, June 2004.

[21] V. Ciaschini, A. Ferraro, A. Ghiselli, G. Rubini, R. Zappi, and A. Caltroni. G-pbox: A Policy Framework for Grid Environments. In *Computing in High Energy and Nuclear Physics*, 2004.

[22] E.G. Coffman. *Computer and Job-Shop Scheduling Theory.* Wiley, 1976.

[23] E.G. Coffman and R.L. Graham. Optimal scheduling for two-processor systems. *Acta Informatica*, 1:200–213, 1972.

[24] M. Consrad, M. Marrakchi, Y. Robert, and D. Trystram. Parallel gaussian elimination on an mimd computer. *Parallel Computing*, 6:275–296, 1988.

[25] K. Cooper, A. Dasgupta, K. Kennedy, C. Koelbel, A. Mandal, G. Marin, M. Mazina, J. Mellor-Crummey, F. Berman, H. Casanova, A. Chien, H. Dail, X. Liu, A. Olugbile, O. Sievert, H. Xia, L. Johnsson, B. Liu, M. Patel, D. Reed, W. Deng, C. Mendes, Z. Shi, A. YarKhan, and J. Dongarra. New grid scheduling and rescheduling methods in the grads project. In *IPDPS Next Generation Software Program - NSFNGS - PI Workshop*, 2004.

[26] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *Proceedings of the 10th IEEE Symposium on High-Performance Distributed Computing*, 2001.

[27] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82, 1998.

[28] H. Dail, F. Berman, and H. Casanova. A decoupled scheduling approach for grid application development environments. *J. Parallel Distrib. Comput.*, 63(5):505–524, 2003.

[29] H. Dail, F. Berman, and H. Casanova. A modular scheduling approach for grid application development environments. *Journal of Parallel and Distributed Computing*, 63(5), 2003.

[30] Holly Dail, Otto Sievert, Fran Berman, Henri Casanova, Asim YarKhan, Sathish Vadhiyar, Jack Dongarra, Chuang Liu, Lingyun Yang, Dave Angulo, and Ian Foster. Scheduling in the grid application development software project. In *Grid Resource Management: State of the Art and Future Trends*, chapter 6, pages 73–98. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[31] A. Darte. Two heuristics for task scheduling, laboratoire lip-imag, ecole normale superieure de lyon, 69364. 1991.

[32] V. A. Dixit-Radiya and D. K. Panda. Task assignment on distributed-memory systems with adaptive wormhole routing. *Proceedings of International Symposium of Parallel and Distributed Systems*, pages 674–681, December 1993.

[33] M. B. Doar. A better model for generating test networks. *IEEE GLOBECOM*, 1996.

[34] EGEE Project. http://public.eu-egee.org/.

[35] H. El-Rewini and T.G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, 9:138–153, 1990.

[36] FAFNER. http://www.npac.syr.edu/factoring.html.

[37] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On powerlaw relationships of the internet topology. In *Proceedings of SIGCOMM*, 1999.

[38] P.C. Fishburn. *Interval Orders and Interval Graphs*. John Wiley & Sons, 1985.

[39] S. Fitzgerald, I. Foster, C. Kesselman, G. Von Laszewski, W. Smith, and S. Tuecke. A directory service for configuring high-performance distributed computations. In *6th IEEE Symposium on High-Performance Distributed Computing*, pages 365–375. IEEE Computer Society Press, August 1997.

[40] A. Forti, S.R. Bavikadi, H. Hornmayer, A. De Angelis, and the MAGIC collaboration. Grid services for the magic experiment. In *Proceedings of the 6th International Symposium "Frontiers of Fundamental and Computational Physics"*, pages 333–337, 2004.

[41] I. Foster, J. Geisler, W. Nickless, W. Smith, and S. Tuecke. Software infrastructure for the i-way high performance distributed computing experiment. In *Proc. 5th IEEE Symposium on High Performance Distributed Computing*, pages 562–571, 1997.

[42] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl. J. Supercomputer Application*, 11(2):115–128, 1997.

[43] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, San Francisco, CA, 1998.

[44] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. *Open Grid Service Infrastructure WG, Global Grid Forum*, June 2002.

[45] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications*, 15(3), 2001.

[46] Ian Foster. The grid: A new infrastructure for 21st century science. *Physics Today*, February 2002.

[47] Ian Foster. What is the grid? a three point checklist. *Grid Today*, 1(6), July 2002.

[48] James Frey, Todd Tannenbaum, Ian Foster, Miron Livny, and Steve Tuecke. Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing*, 5:237–246, 2002.

[49] D.K. Friesen. Tighter bounds for lpt scheduling on uniform processors. *SIAM Journal on Computing*, 16(3):554–560, June 1987.

[50] M.R. Garey, D. Johnson, R. Tarjan, and M. Yannakakis. Scheduling opposing forests. *SIAM Journal on Algebraic Discrete Methods*, 4(1):72–92, 1983.

[51] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, 1979.

[52] A. Gerasoulis and S. Venugopal. Linear clustering of linear algebra task graphs for local memory systems. Technical report.

[53] A. Gerasoulis and T. Yang. A comparison of clustering heuristics for scheduling directed directed acyclic graphs on multiprocessors. *Journal of Parallel and Distributed Computing*, 16:276–291, 1992.

[54] A. Gerasoulis and T. Yang. On the granularity and clustering of directed acyclic task graphs. *IEEE Transactions on Parallel and Distributed Systems*, 4:686–701, June 1993.

[55] C.A. Glass, C.N. Potts, and P. Shade. Unrelated parallel machine scheduling using local search. *Mathematical and Computer Modelling*, 20(2):41–52, July 1994.

[56] M.J. Gonzalez. Deterministic processor scheduling. *ACM Computing Surveys*, 9(3):173–204, September 1977.

[57] R.L. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.*, 17:416–429, 1969.

[58] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnoy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, (5):287–326, 1979.

[59] A. Grimshaw and et al. The legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1):39–45, 1997.

[60] T.C. Hu. Parallel sequencing and assembly line problems. *Oper. Research*, 19(6):841–848, November 1961.

[61] J. J. Hwang, Y. C. Chow, F. D. Anger, and C. Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, 18(2):244–257, April 1989.

[62] Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw Hill, 1993.

[63] O. Ibarra and C. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *Journal of the ACM*, 77(2):280–289, April 1977.

[64] M. A. Iverson, F. Ozguner, and G. J. Follen. Parallelizing existing applications in a distributed heterogeneous environment. *4th Heterogeneous Computing Workshop (HCW'95)*, pages 93–100, April 1995.

[65] Aubin Jarry, Henri Casanova, and Francine Berman. Dagsim: A simulator for dag scheduling algorithms. Technical Report RR2000-46, LIP, 2000.

[66] D. Kim and B.G. Yi. A two-pass scheduling algorithm for parallel programs. *Parallel Computing*, 20:869–885, 1994.

[67] S. J. Kim and J. C. Browne. A general approach to mapping of parallel computation upon multiprocessor architectures. *Proc. Int'l. Conf. on Parallel Processing*, pages 1–8, 1998.

[68] W.H. Kohler and K. Steiglitz. Characterization and theoretical comparison of branch-and-bound algorithms for permutation problems. *Journal of the ACM*, 21(1):140–156, January 1974.

[69] S. Konya and T. Satoh. Task scheduling on a hypercube with link contentions. *Proceedings of International Parallel Processing Symposium*, pages 363–368, April 1993.

[70] H. Kornmayer, M. Hardt, M. Kunze, C. Bigongiari, M. Mazzucato, A. De Angelis, G. Cabras, A. Forti, M. Frailis, M. Piraccini, and M. Delfino. A distributed, grid-based analysis system for the magic telescope. In *CHEP04*, 2004.

[71] B. Kruatrachue and T.G. Lewis. Duplication scheduling heuristics (dsh): A new precedence task scheduler for parallel processor systems. Technical report, Oregon State University, Corvallis, OR 97331, 1987.

[72] P. E. Krueger. Distributed scheduling for a changing environment. Technical Report UW-CS-TR-780, Universtity of Wisconsin, Computer Science Department, June 1988.

[73] Y. K. Kwok. Efficient algorithms for scheduling and mapping of parallel programs on parallel architectures. Master's thesis, HKUST, Hong Kong, 1994.

[74] Y. K. Kwok and I. Ahmad. A static scheduling algorithm using dynamic critical path for assigning parallel algorithms onto multiprocessors. *Proceedings of International Conference on Parallel Processing*, 2:155–159, August 1994.

[75] Y. K. Kwok, I. Ahmad, and J. Gu. Fast: A low-complexity algorithm for efficient scheduling of dags on parallel processors. *Proceedings of 25th International Conference on Parallel Processing*, 2:150–157, August 1996.

[76] Yu-Kwong Kwok. *High-performance algorithms of compile-time scheduling of parallel processors.* PhD thesis, Hong Kong University of Science and Technology, 1997.

[77] Yu-Kwong Kwok and I. Ahmad. Bubble scheduling: A quasi dynamic algorithm for static allocation of tasks to parallel architectures. In *Proceedings of the 7th IEEE Symposium on Parallel and Distributeed Processing*, page 36, Washington, DC, USA, 1995. IEEE Computer Society.

[78] Yu-Kwong Kwok and Ishfaq Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):506–521, 1996.

[79] Yu-Kwong Kwok and Ishfaq Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59:381–422, 1999.

[80] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, 1999.

[81] A. Legrand, L. Marchal, and H. Casanova. Scheduling distributed applications: the simgrid simulation framework. In *Proc. of the 3rd International Symposium on Cluster Computing and the Grid*, pages 138–145, 2003.

[82] Arnaud Legrand and Julien Lerouge. Metasimgrid : Towards realistic scheduling simulation of distributed applications. Technical report, LIP, July 2002.

[83] H. Levy and E. Tempero. Modules, objects and distributed programming: Issues in rpc and remote object invocation. *Software Practice and Experience*, 21(1):77–90, January 1991.

[84] T.G. Lewis and H. El-Rewini. *Introduction to Parallel Computing*. Prentice-Hall, 1992.

[85] Jing-Chiou Liou and Michael A. Palis. An efficient task clustering heuristic for scheduling dags on multiprocessors. In *Symposium of Parallel and Distributed Processing*, 1996.

[86] M. Litzkow, M. Livny, and M. Mutka. Condor - A hunter of idle workstations. In *8th International Conference of Distributed Computing Systems (ICDCS)*, pages 104–111, Los Alamitos, CA, USA, June 1988. IEEE Computer Society Press.

[87] Eckart Lorenz and the MAGIC collaboration. Status of the 17m diameter magic telescope. *New Astronomy Reviews*, 48(5-6):339–344, April 2004.

[88] Dong Lu and Peter A. Dinda. Gridg: Generating realistic computational grids. *Performance Evaluation Review*, 30(4):33–40, March 2003.

[89] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *8th Heterogeneous Computing Workshop (HCW'99)*. IEEE Computer Society, April 12 1999.

[90] M. Maheswaran and H. J. Siegel. A dynamic matching and scheduling algorithm for heterogeneous computing system. In *the 7th Heterogeneous Computing Workshop(HCW '98)*, pages 57–69. IEEE Computer Society Press, March 1998.

[91] Dan C. Marinescu. A Grid Workflow Management Architecture. GGF White Paper, 2002.

[92] Dan C. Marinescu. *Internet-Based Workflow Management: Toward a Semantic Web.* Wiley, 2002.

[93] C. McCreary and H. Gill. Automatic determination of grain size of efficient parallel processing. *Communications of ACM*, 32(9):1073–1078, September 1989.

[94] N. Mehdiratta and K. Ghose. A bottom-up approach to task scheduling on distributed memory multiprocessor. *Proceedings of International Conference on Parallel Processing*, 2:151–154, August 1994.

[95] Distributed Net. http://www.distributed.net/.

[96] B. Neuman. Scale in distributed systems. In T. Casavant and M. Singhal, editors, *Readings in Distributed Computing Systems*, pages 463–489. IEEE Computer Society Press, 1994.

[97] M.A. Palis, J.-C. Liou, and D.S.L. Wei. Task clustering and scheduling for distributed memory parallel architectures. *IEEE Transactions on Parallel and Distributed Systems*, 7(1):46–55, January 1996.

[98] Santosh Pande, Dharma P. Agrawal, and Jon Mauney. A scalable scheduling scheme for functional parallelism on distributed memory multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(4):388–399, April 1995.

[99] C.H. Papadimitriou and M. Yannakakis. Scheduling interval-ordered tasks. *SIAM J. Computing*, 8:405–409, 1979.

[100] C.H. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM Journal on Computing*, 19(2):18–29, April 1990.

[101] G.L. Park, B. Shirazi, and J. Marquis. Dfrn: A new approach for duplication based scheduling for distributed memory multiprocessor systems. *Proc. 11th Inter. Parallel Processing Symposium*, pages 157–166, April 1997.

[102] Rosario M. Piro, Andrea Guarise, and Albert Werbrouck. An Economy-based Accounting Infrastructure for the DataGrid. In *Proceedings of the 4th International Workshop on Grid Computing (GRID2003)*, 2003.

[103] M.J. Quinn. *Parallel Computing: Theory and Practice.* McGraw-Hill, 1994.

[104] A. Radulescu and A. J. C. van Gemund. Flb: Fast load balancing for distributed-memory machines. In *Proc. Int'l Conf. on Parallel Processing*, 1999.

[105] Andrei Radulescu and Arjan J.C. van Gemund. On the complexity of list scheduling algorithms for distributed-memory systems. In ACM Press, editor, *ICS '99: Proceedings of the 13th international conference on Supercomputing*, pages 68–75, New York, NY, USA, 1999.

[106] David De Roure, Mark A. Baker, Nicholas R. Jennings, and Nigel R. Shadbolt. The evolution of the grid. In *Grid Computing - Making the Global Infrastructure a Reality*, pages 65–100. John Wiley and Sons Ltd, 2003.

[107] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors.* The MIT Press, 1989.

[108] J. Schopf and F. Berman. Stochastic scheduling. In *Proceedings of Super Computing (SC99)*, 1999.

[109] Jennifer M. Schopf. Ten actions when grid scheduling. In *Grid resource management: state of the art and future trends*, chapter 2, pages 15–23. Kluwer Academic Publishers, 2003.

[110] S. Selvakumar and C.S.R. Murthy. Scheduling precedence constrained task graphs with non-negligible intertask communication onto multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):328–336, March 1994.

[111] R. Sethi. Scheduling graphs on two processors. *SIAM Journal of Computing*, 5(1):73–82, March 1976.

[112] SETI@Home. http://setiathome.ssl.berkeley.edu/.

[113] Gary Shao, Francine Berman, and Richard Wolski. Using effective network views to promote distributed application performance. In *PDPTA*, pages 2649–2656, 1999.

[114] Howard Jay Siegel and Shoukat Ali. Techniques for mapping tasks to machines in heterogeneous computing systems. *J. Syst. Archit.*, 46(8):627–639, 2000.

[115] Gilbert C. Sih and Edward A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Trans. on Parallel Distributed Systems*, 4(2):175–187, 1993.

[116] M. Srinivas and L.M. Patnaik. Adaptive probabilities of crossover and mutation in genetic algorithms. *IEEE Transactions on Systems, Man and Cybernetics*, 24(4):656–667, April 1994.

[117] H. Stone. *High-Performance Computer Architectures.* Reading, Mass.:Addison-Wesley, 1987.

[118] Andrew S. Tanenbaum and Maarten van Steen. *Distributed systems : principles and paradigms.* 2002.

[119] Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny. Condor – A distributed job scheduler. In Thomas Sterling, editor, *Beowulf Cluster Computing with Linux.* MIT Press, October 2001.

[120] Douglas Thain, Todd Tannenbaum, and Miron Livny. Condor and the grid. In Fran Berman, Geoffrey Fox, and Tony Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*, chapter 11. John Wiley & Sons Inc., December 2002.

[121] The DataGrid Project. http://eu-datagrid.web.cern.ch/.

[122] W3C The World Wide Web Consortium. http://www.w3.org/.

[123] B. Tierney, R. Aydt, D. Gunter, W. Smith, M. Swany, V. Taylor, and R. Wolski. A grid monitoring architecture. Technical report GFD-I.7, Global Grid Forum (GGF), 2003.

[124] J. Ullman. NP-Complete Scheduling Problems. *Journal of Computer and System Sciences*, 10:384–393, 1975.

[125] UNICORE Forum. UNICORE Plus Final Report: Uniform Interface to Computing Resource. 2003, http://www.unicore.org/documents/UNICOREPlus-Final-Report.pdf [December 2004].

[126] L. Wang, H. J. Siegel, V. P. Roychowdhury, and A. A. Maciejewski. Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. *Joural of Parallel and Distributed Computing*, 47(1):8–22, November 1997.

[127] M.-F. Wang. Message routing algorithms for static task scheduling. *Proceedings of the 1990 Symposium on Applied Computing*, pages 276–281, 1990.

[128] V. Welch, I. Foster, C. Kesselman, O. Mulmo, L. Pearlman, S. Tuecke, J. Gawor, S. Meder, and F. Siebenlist. X.509 proxy certificates for dynamic delegation. In *3rd Annual PKI R&D Workshop*, 2004.

[129] V. Welch, F. Siebenlist, I. Foster, J. Bresnahan, K. Czajkowski, J. Gawor, C. Kesselman, S. Meder, L. Pearlman, and S. Tuecke. Security for grid services. *Twelfth International Symposium on High Performance Distributed Computing (HPDC-12)*, June 2003.

[130] Rich Wolski, Neil Spring, and Jim Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems*, 15(5-6):757 – 768, October 1999.

[131] W.S. Wong and R.J.T. Morris. A new approach to choosing initial points in local search. *Information Processing Letters*, 30(2):67–72, January 1989.

[132] Min-You Wu and D. Gajski. A programming aid for hypercube architectures. *The journal of Supercomputing*, 2:349–372, 1988.

[133] Min-You Wu and Daniel D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):330–343, July 1990.

[134] T. Yang and A. Gerasoulis. List scheduling with and without communication delays. *Parallel Computing*, 19(12):1321–1344, 1993.

[135] T. Yang and A. Gerasoulis. Dsc: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, September 1994.

[136] Jia Yu and Rajkumar Buyya. A taxonomy of scientific workflow systems for grid computing. *SIGMOD Rec.*, 34(3):44–49, 2005.