Alexandru Jugravu

A High-level Programming Paradigm for Java-based Parallel and Distributed Applications

8. September 2005

TECHNISCHE
UNIVERSITÄT
WIEN

VIENNA
UNIVERSITY OF
TECHNOLOGY

D I S S E R T A T I O N

A High-level Programming Paradigm for Java-based Parallel and Distributed Applications

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Doktors der technischen Wissenschaften unter der Leitung von

o. Univ.-Prof. Dipl. Ing. Dr. Thomas Fahringer
Institut für Informatik, Leopold-Franzens Universität Innsbruck

eingereicht an der Technischen Universität Wien
Fakultät für Informatik

von

Dipl. Ing. Alexandru Jugravu
Matrikelnummer: 0127156
Liechtensteinstr. 22A/2/2/18, A-1090 Wien

Wien, am 8. September 2005

Dedicated to
My parents,
Amelia, and
Ana Maria

Abstract

With the development of the Internet and Web computing, the sharing of programs across heterogeneous platforms and the establishment of a unified programming and computing environment across the fundamentally heterogeneous World Wide Web have become critical issues, which led to the booming of brand new programming languages such as Java.

There has been an increasing research interest in extending the use of Java towards performance-oriented programming for distributed and concurrent applications. Numerous research projects have introduced class libraries or language extensions for Java, and tried to provide flexible and high-level APIs for programming parallel and distributed applications. Much of this work focuses on providing automatic performance control (e.g. automatic load balancing, mapping, migration, or control of locality) for distributed applications and assumes that the runtime system is able to detect parallelism, exploit locality and achieve efficient load balancing. However, automatic load balancing and data migration can easily lead to significant performance degradation, as the underlying runtime system is lacking sufficient information about the distributed applications. In many cases, programmers are very much aware of the particular nature of their application, how to distribute data, which data should be mapped together, when to migrate data, etc. Programming paradigms that disable the programmer to provide the runtime system with this information may severely degrade performance.

To overcome the limitations of existing programming paradigms, this thesis proposes JavaSymphony, a novel programming paradigm for wide classes of heterogeneous systems ranging from small-scale cluster computing to large scale wide area meta computing. On the one hand, JavaSymphony supports automatic mapping, load balancing and migration of objects without involving the programmer. On the other hand, in order to enhance the performance of distributed applications, JavaSymphony provides a semi-automatic mode, which leaves the error-prone, tedious, and time consuming low-level details to the underlying system, whereas the programmer controls the most important strategic decisions at a very high level.

We have designed JavaSymphony Runtime System as a distributed Java-based middleware to support the execution of distributed JavaSymphony applications. The components of the JavaSymphony middleware (called agents) are running onto distributed resources and provide basic services needed by the applications like communication, resource and application monitoring, or code execution. This dissertation presents in detail relevant features of the JavaSymphony Runtime System design. Furthermore, we propose a formal representation to describe middleware functionality, based on the Pi-calculus.

The JavaSymphony programming paradigm allows flexible implementation of a large range of distributed applications, including workflow applications, which are quite suitable for coarse-grain distributed computing. On the other hand, in many cases, the improvement of application performance may require significant programming effort, whilst the applications that adhere to a specific computation/communication model such as a workflow model, allow performance tuning through automatic scheduling and resource allocation. Motivated by these aspects, we introduce a framework for scheduling workflow applications in JavaSymphony. Our approach in this matter differs from similar research in several ways: Whilst in most related work, the workflows are limited to DAGs of tasks, we present a workflow model that includes loops to model repetition in workflow applications, and conditional branches to address non-deterministic behaviour due to data that is available only at runtime. Furthermore, we build a dynamic scheduling strategy that addresses the new workflow elements and we apply this technique to enhance several static DAG-based scheduling heuristics. In addition, the thesis introduces a theoretical framework to describe the functionality of the resource broker, which support advanced features like reservations and dynamic updates of the estimated task execution times.

Finally, the thesis presents a variety of real-world experiments that validate the research topics addressed.

Acknowledgements

Here I would like to acknowledge my gratitude to some people who helped me in completing this work.

First of all, I would like to specially thank my advisor, Prof. Thomas Fahringer, for his continued generous support and guidance during my studies, and for the opportunity of working in his research group at University of Vienna. Without his extremely valuable assistance, this thesis would likely not have matured.

I am also deeply thankful to Prof. Schahram Dustdar for accepting to be my second co-examiner.

I thank Prof. Zima for giving me the opportunity to work in Aurora research program, which has funded my Ph.D. studies, and to the members of the AURORA project for their cooperation, which had a deep impact on the outcome of my work.

I would also like to thank my colleagues in the tool group, Radu Prodan, Lihn Truong, Clovis Seragiotto and Sabri Pllana, for many valuable discussions on research or social issues. I wish to address special thanks to Radu Prodan, for partially proofreading this thesis and for his helpful comments.

Finally, my warm thanks go to my beloved wife, Amelia, and my parents, for their continuous support and encouragement, and to my wonderful daughter, Ana Maria, for bringing joy in my life.

Contents

List of Figures

List of Tables

1 Introduction

In the last decades, we have witnessed a revolution of the computing systems, from massive and largely expensive computers introduced in the 1940s, to parallel and distributed architectures comprising large numbers of powerful microprocessors connected by high-speed networks. Nowadays, distributed systems have become the norm for the organization of computing facilities. The availability of high-performance personal computers, workstations, and server computers has resulted in a major shift towards distributed systems and away from centralized and multi-user computers. Clusters of inexpensive workstations or multi-processors are increasingly popular as a promising solution to design low cost parallel machines.

In parallel with the rapid advances in computing architecture and technology, we have seen a growing demand for computation power from increasingly complex applications. It seems that the size and complexity of the problems we would like to deal with grow faster than the development of hardware technologies. The demand for even greater application performance is a familiar feature of every aspect of computing. Advances in hardware capability enable new application functionality, which grows in significance and places even greater demands on the architecture. Application demand for computational performance continues to outpace what individual processor can deliver, and therefore multiprocessor systems occupy an increasingly important place in mainstream computing.

On the other hand, efficient utilization of the huge computing power available in single supercomputers, or as the sum of the computing power of the components of a distributed system (e.g. a cluster of workstations, a geographically distributed grid of resources, or even all the hosts available over the Internet) is a difficult task. Motivated by this aspect, considerable efforts have been also made in developing software that can efficiently use the new computing architectures.

Presently, Local Area Networks (LANs) are able to link hundreds of machines within a building, whilst Wide Area Networks (WANs) allows millions of machines all over the earth to be connected. It is very attractive to harvest the cycles of the interconnected idle machines distributed all over the world, which can provide peta-flops of aggregated computing power. But beyond the evident advantages of using already existing hardware and a very competitive cost to performance ratio, there are also drawbacks of using heterogeneous systems such as high communication overhead, or compatibility issues, due to the variety of the hardware, operating systems or the software used.

We have identified several major advances in the software science that have encouraged the shift towards heterogeneous computing, as a viable alternative to the massively parallel architectures.

The Internet and the World Wide Web

The Internet [3] was invented in the 1980s, but has exploded in popularity on a worldwide scale with the advent of the World Wide Web (WWW) [4] later in the 1990s. Despite its global success and acceptance as a standard mean of publishing and exchange of digital information, the WWW technology does not enable ubiquitous access to the billions of (potentially idle) computers simultaneously connected to the Internet providing huge amount of estimated aggregate computational power.

Java

With the development of the Internet and Web computing, the sharing of programs across heterogeneous platforms and the establishment of a unified programming and computing environment across the fundamentally heterogeneous WWW have become critical issues, which led to the booming of brand new programming languages such as Java [5, 6]. Java may be not distinct in its way of programming, but it is definitely distinguished in the execution of its programs on various hardware platforms, available at the present or which may not even exist right now.

Since its introduction in May 1995, the Java platform has been adopted more quickly across the industry than any other new technology in computing history. All major computing platform vendors have signed up to integrate Java technology as a core component of their products. The popularity and the utilization of the Java technology have grown over the past ten years, because of Java's true portability. The Java platform enables the same Java application run on most existing machines, no matter which computing architecture or which operating system they use. Moreover, Java offers build-in support for code-mobility, object-orientation, multi-threading, security, remote communication, which are highly useful for distributed computing.

On the other hand, Java programs are commonly interpreted, and often use more memory than programs written in lower-level language. Moreover, automatic garbage collection and array range checking can seriously affect the performance, especially for applications that run only for short time. However, in the last couple of years, several efforts have been made to solve Java's performance problems, such as just-in-time compilation or dynamic recompilation (which allow the program to take advantage of the speed of native code without loosing the portability), and many optimisations in the Java Virtual Machine (JVM). Recent results [7] have shown that optimised Java code can perform comparably to C or Fortran for specific classes of applications. Still, whether Java is significantly slower than other languages is hotly debated.

Grid computing

Analogous to the World Wide Web that provides ubiquitous access to the information on the Internet, the computational Grids explore new mechanisms for ubiquitous access to computational resources and quality of service beyond the best-effort provided by the Internet Protocol (IP).

Built on pervasive Internet standards, Grid computing [8] enables organizations to share computing and information resources across department and organizational boundaries in a secure, highly efficient manner. The Grid computing is commonly illustrated by an analogy with an electrical power grid [8, 9]. The users of the Grid should be able to access many and diverse resources (e.g. high-end computational capabilities, aggregated computing power of many idle CPUs or large collection of storage/data) as simply as plugging a device into an electrical socket.

The Grid research challenge is to provide standard, reliable, and low-cost access to the relatively cheap computing power available nowadays [10]. Grid computing involves sharing heterogeneous resources over a network by using open standards. The resources may be based on different platforms, hardware/software architectures, and computer languages, and may belong to different administrative domains. The ensemble of resources is meant to support the execution of large-scale, resource-intensive, and distributed applications.

Nowadays, there are many different points of view about what Grid computing means, and there are still many open issues regarding the Grid computing, which are currently addressed by numerous research groups. In order to realize the benefits of Grid computing, standards are needed, so that the diverse resources can be discovered, accessed, allocated, monitored, and in general managed as a single virtual system, even when provided by different organizations. Currently, the Open Grid Service Architecture (OGSA) [11] and the Globus Toolkit [12, 13] are playing a major role in providing the needed Grid computing standards.

1.1 Motivation

In the past years, the interest in computational Grids has constantly grown in the scientific community as a mean of enabling application developers to aggregate the capabilities of heterogeneous resources scattered around the globe for solving large-scale scientific problems. Developing applications that can effectively utilise the huge amount of the resources remains, however, a very difficult task, because of the lack of high-level programming paradigms and tools to support developers.

This thesis aims to meet various aspects concerning the programming of distributed applications for heterogeneous distributed environments.

1.1.1 Programming Paradigms for Heterogeneous Distributed Architectures

Programming for parallel and distributed architectures is much more complex than for sequential computers. Typically, the programmer is provided with a machine-specific low-level programming interface (e.g. OpenMP [14] for shared memory multi-processors, MPI [15] for distributed memory multiprocessors system, or hybrid OpenMP/MPI for SMP clusters). Moreover, on top of low-level message passing, distributed systems like Sun *Remote Procedure Call* (RPC) [16], or Microsoft's *Distributed Computing Environment* (DCE) [17], enable the execution of code via remote procedure calls. The idea of distributed objects extends the concept of remote procedure calls with object-oriented programming, and has been used in popular distributed programming paradigms like Microsoft's *Distributed Component Object Model (DCOM)* [18], *Common Object Request Broker Architecture (CORBA)* [19], designed by Object Management Group (OMG), and JavaSoft's *Java/Remote Method Invocation (Java/RMI)* [20]. All these three standards have their component model extensions.

However, the existing tools and programming models are still too low-level and developing distributed applications commonly requires to deal with all the details of the parallelism (e.g. decomposition of the program and data, the mapping of the subparts to the machines/processors, communication and synchronization). The developer cannot avoid managing the low-level programming aspects like explicitly building distinct threads of execution, identifying the available/suitable resources and mapping code and data onto them. Therefore, new programming paradigms that can support the easy development of parallel and distributed applications are in great demand.

1.1.2 Portability

Heterogeneous resources usually have not only various architectures and computing powers, but also distinct operating systems and distinct software installed. Portability is necessary to guarantee that these resources can collaborate by using a unified programming paradigm or middleware. Building a portable programming paradigm on top of Java, a true portable programming language, is therefore a desirable choice.

1.1.3 Performance

Achieving performance on a heterogeneous computing infrastructure is not an easy task. Heterogeneous computing systems commonly consist of a large number of machines interconnected through off-the-shelf communication components, or even by a WAN. This implies higher latency and lower bandwidth that produce scalability problems, which make the heterogeneous computing not suitable for several classes of applications (e.g. application with low computation to communication ratios).

On the other hand, other distributed systems properties, such as flexibility, portability and transparency, usually come at the cost of performance.

Performance versus portability. In order to run parts of a distributed application on various computing architectures, potentially having distinct operating systems and distinct software installed, a new middleware layer is required to hide the differences between them. For example, Java programs need a Java Virtual Machine (JVM) to run on the target architecture. The additional software layer causes overhead and thus affects the performance of the distributed applications.

Performance versus transparency. Distributed systems commonly hide the details of the underlying resources (e.g. location, performance parameters, architecture, etc.), in order to facilitate their usage by various types of distributed applications. We believe that the access to certain performance information about the underlying resources may be useful in order to improve the performance of certain applications. This information could be used automatically by an application scheduler for example, or may be accessed within the application by using a high-level API to system parameters.

Automatic versus user-controlled performance tuning. Many research groups have tried to provide flexible and high-level APIs for programming parallel and distributed applications. Much of this work focuses on providing automatic performance control (e.g. automatic load balancing, mapping, migration, or control of locality) for distributed applications and assumes that the runtime system is able to detect parallelism, exploit locality and achieve efficient load balancing. However, automatic load balancing and data migration can easily lead to significant performance degradation as the underlying runtime system is lacking sufficient information about the distributed applications. In many cases, programmers are very much aware of the particular nature of their application, how to distribute data, which data should be mapped together, when to migrate data, etc. Programming paradigms that disable the programmer to provide the runtime system with this information may severely degrade performance. Therefore, we believe that a programming paradigm should support both options: automatic or user-controlled performance tuning in regard with mapping, migration and load balancing.

1.1.4 Access and Control of Resources

High-level programming paradigms for heterogeneous distributed systems are required to provide APIs for the management of the distributed resources. The programming paradigm is supposed to hide the low-level details regarding the location of the resources and/or the protocols used to access them. However, the programmers may require information from the runtime system to support or to refine their strategic decisions. Thus, a high-level API to system parameters is needed to mitigate the programming effort. Clearly, a programming paradigm should be provided to support and enhance the interplay between programmer and runtime system to greatly improve efficiency and scalability of programs targeting modern parallel and distributed computing structures, while simplifying the programming effort.

1.1.5 Fault Tolerance and Adaptation to Dynamic Changes of Distributed Systems

Heterogeneous systems, and in particular Grids, are characterized by dynamic changes regarding the availability or the performance properties of the underlying machines. The resources may be shared by multiple applications, and therefore the performance could vary substantially. On the other hand, resource may randomly fail or become available (again). Resource monitoring is required to determine when such events happen, and support for migration of code and data may be needed if these events occur. Many distributed systems commonly do not address the failures or the dynamic changes in performance of the resources.

1.1.6 Scheduling

Many distributed applications follow a well-defined pattern, which allows performance optimization by appropriate mapping and synchronization between the concurrent components of the application. These components could be automatically scheduled onto the resources, thus saving significant programming effort.

Application scheduling in a classical approach is an NP-complete optimisation problem [21]. The scheduling algorithms search within a space that exponentially grows with the (potentially unbounded) number of resources and tasks and that can achieve particularly huge dimensions on the Grid, which have not been previously addressed. In addition, the static scheduling as an optimisation problem has to be enhanced with steering capabilities that consider the dynamic availability of the resources over space and time. The workflow model originating from business process modelling [22] has gained increased interest as the potential state-of-the-art paradigm for programming distributed applications. While business process workflows are in most cases Directed Acyclic Graphs (DAG) that consist of a limited number of nodes, scientific workflows that implement Grid applications often require large iterative loops that model a convergent behaviour or a recursive problem definition, or employ selection criteria that produce dynamic changes in the structure of their execution graphs.

1.1.7 Formal Representation of Distributed Processes

We have observed that many research projects for distributed and heterogeneous computing focus on implementation issues and lack a theoretical approach. We believe that a formal representation can be used to model the distributed processes and may support an in-depth theoretical understanding of the model, reasoning and verification.

1.2 Goals

Motivated by the problems outlined in the previous section, we aim to support the developer of distributed applications with an easy to use novel high-level programming paradigm for performance oriented parallel and distributed applications.

1.2.1 High-level Programming Paradigm for Distributed Heterogeneous Systems

In this thesis, we propose JavaSymphony, a novel programming paradigm for wide classes of heterogeneous systems ranging from small-scale cluster computing to large scale wide area meta computing. JavaSymphony supports object-oriented distributed computing and it is particularly well-suited for applications that require shared address space, task parallelism, or one-side message passing.

The primary goal of the JavaSymphony programming paradigm is to alleviate the development of parallel and distributed Java programs. At the same time, the improvement of the application performance is playing an important role. For performance reasons, the JavaSymphony programming paradigm strongly supports the programmer to specify and to control locality, parallelism, and load balancing at a high level, without putting a burden on the programmer to deal with error-prone and time consuming low-level details (e.g. socket communication or creation and management of remote proxies for Java/RMI mechanism).

In addition, JavaSymphony provides programming elements highly useful for programming distributed applications that run on heterogeneous systems, which includes:

- High-level control over distributed resources;

- High level access to a large variety of static or dynamic system parameters, including machine name, user name, operating system, JVM version or CPU load, idle time, available memory size, number of processes and threads, network latency, network bandwidth, etc;
- Selective remote class-loading;
- Automatic and user-controlled mapping and migration of distributed objects;
- Multi- and single-threaded access to distributed objects;
- Communication through several types of remote method invocation;
- Distributed synchronization mechanisms;
- Distributed event mechanism;

Moreover, the JavaSymphony programming paradigm is characterized by several desirable properties outlined in Section 1.1:

Portability. Portability is ensured by the fact that JavaSymphony provides a class library, which is entirely written in Java and runs on any standard compliant JVM. There is no need to invest time in learning new languages from scratch, since the supported distributed applications are written in Java by only using the JavaSymphony class library.

Performance. On the one hand, JavaSymphony supports automatic mapping, load balancing and migration of objects without involving the programmer. On the other hand, in order to enhance the performance of distributed applications, JavaSymphony provides a semi-automatic mode, which leaves the error-prone, tedious, and time consuming low-level details (e.g. creating and handling of remote proxies for Java/RMI) to the underlying system, whereas the programmer controls the most important strategic decisions at a very high level.

Resource Management. JavaSymphony introduces the concept of *dynamic virtual distributed architectures*, which allows the programmer to define a structure of a heterogeneous (e.g. type, speed, or configuration) network of computing resources and to control the code placement and the mapping, load balancing, and migration of objects. Moreover, it gives the programmer the opportunity to access a large set of dynamic and static system parameters, in order to improve the performance of the application.

1.2.2 JavaSymphony - Middleware for Java-based Distributed Applications

The distributed applications use the JavaSymphony programming paradigm in order to run their components on multiple computing resources. We have implemented the JavaSymphony Runtime System, a middleware that is needed to hide the low-level details of the operating systems and to offer a higher level of abstraction to JavaSymphony application developers. The components of the JavaSymphony Runtime System (called agents) are running onto distributed resources and provide basic services needed by the applications (e.g. communication, resource and application monitoring, code execution, etc.). In this thesis, we present in detail relevant features of the JavaSymphony middleware.

1.2.3 Scheduling Task-based Distributed Applications

The workflow model has emerged as a very promising paradigm for programming distributed applications. Commonly, a static scheduling strategy is used to build a schedule for a DAG-based workflow, which is known as an NP-complete optimisation problem. However, static scheduling is not appropriate for dynamic distributed environments, in which resources may randomly become unavailable or unsuitable and may change dramatically their performance parameters. At the same time, DAG-based workflow models cannot express dynamic behaviour that can occur due to criteria that change at runtime or due to repetition until convergence criteria are met. We propose a new workflow model, which includes loops and conditional branches to address these issues. For workflow applications of this type, we introduce a novel dynamic scheduling method.

1.2.4 A Formal Model for Distributed Systems

This thesis proposes a theoretical framework to describe distributed systems, based on a formal language. For this purpose, we have chosen the Pi-calculus, which can be seen as a minimal programming language built to capture all interesting behaviours of concurrent programs and which gives us a mean of expressing the dynamic interactions among communicating processes. We have extended this calculus in order to make it suitable for the formal representation of concurrent and distributed processes in two cases:

- Firstly, we investigate how to represent the functionality of the JavaSymphony Runtime System as an abstract Pi-calculus process.
- Secondly, the calculus is used to describe the elements of workflow applications and model the interactions between them.

1.3 Thesis Outline

The rest of the thesis is organized as follows:

Chapter 2 introduces preliminary notions on top of which the concepts presented in this thesis are developed. We discuss the existing computing architectures and the types of applications that can use them.

Chapter 3 explains the JavaSymphony programming paradigm and give details about the JavaSymphony programming constructs.

Chapter 4 presents the functionality of the JavaSymphony middleware and the relevant implementation details. Additionally, a formal model is used to describe the JavaSymphony Runtime System.

Chapter 5 introduces a framework for scheduling workflow applications. Novel scheduling techniques are introduced to manage loops and conditional branches in workflows. A framework for modelling the resource broker is presented in Chapter 6.

Chapter 7 illustrates practical experiments that validate our techniques and evaluate the performance of the JavaSymphony middleware.

Chapter 8 outlines the most relevant related work.

Finally, Chapter 9 summarizes the contributions and outlines future work.

2

Model

Before starting to describe in detail the main topics of the thesis, we introduce a few preliminaries. The first part of the chapter discusses on the available computing systems, ranging from single computing resource to a large scale Grid architecture. We study the computer organizations and classify them according to Flynn's taxonomy [23]. Secondly, we investigate the applications that can use the described computing architectures, which comprise single address space applications, a variety of distributed applications (e.g. generic collaborative distributed applications, meta-tasks, and workflows), and several types of Grid applications.

2.1 Architectural Model

2.1.1 Single Computing Resource

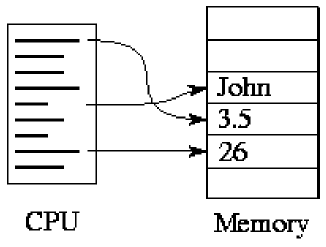


Fig. 2.1. Von Neumann architecture [1]

Most of the past and present computers are based on the single machine model, called von Neumann architecture. A von Neumann computer comprises a single CPU (central processing unit) connected to a single storage structure, which holds both the set of instructions on how to perform the computation and the data produced or required by the computation (Fig. 2.1). The unique CPU can execute only one stream of instructions and therefore it supports only the *SISD (Single Instruction Single Data)* programming model in Flynn's classification [23].

2.1.2 Symmetric Multi-Processor Machine

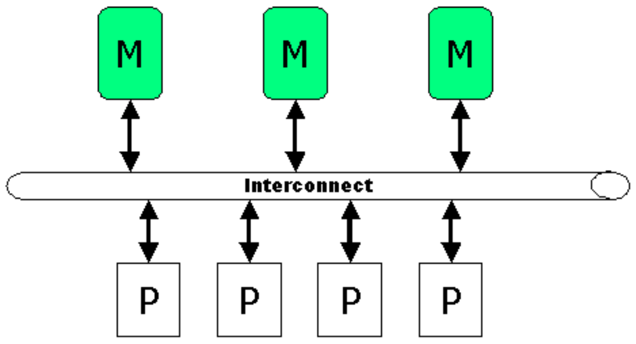


Fig. 2.2. Multiprocessor architecture

A **Symmetric Multi-Processor (SMP) machine** or **an SMP (computing) node** has two or more identical CPUs that are connected to a common memory (consisting of one or more memory modules), commonly via a shared bus (Fig. 2.2). The cost of accessing the shared memory is the same for all CPUs; however, each CPU may have its local cache. The SMP architecture supports *SIMD (Single Instruction, Multiple Data)* and *MIMD (Multiple Instructions, Multiple data*) programming models in Flynn's taxonomy. One important aspect is that a SMP machine uses a single operating system and all the CPUs share the same input/output resources.

The SMP architectures are considered to be **multiprocessors**, which are characterized by the fact that all their CPUs have direct access to a shared memory. Multiprocessors are also denoted as **MIMD shared-address-space** computers [24]. Other examples of multiprocessors are the **NUMA (NonUniform Memory Access) machines**. In this case, in contrast to the SMP architecture, each CPU has access to local, respectively to non-local memory, both shared, and the access to the local memory is faster. Commonly, the present multiprocessor systems use the SMP architecture, since this provides high throughput and performance through multiprocessing, and it is relatively straightforward to develop parallel programs for it. On the other hand, a major disadvantage of the SMP systems is that they provide only limited scalability.

2.1.3 Multicomputers

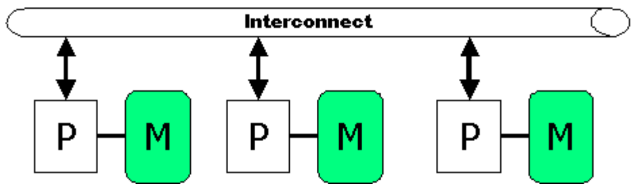


Fig. 2.3. Multicomputer architecture

According to Foster [1], **a multicomputer** comprises a number of von Neumann computers, or nodes, linked by an interconnection network (Fig. 2.3).

The main characteristic of a multicomputer is that, in contrast to the multiprocessors, each CPU has a direct connection to its own private memory, whilst the access to the memory of other CPUs is significantly more expensive and requires specific communication protocols. On the other hand, multicomputer systems are highly scalable, this being their main advantage in comparison with the multiprocessors: Thousands of CPUs can be connected through a single, often high-performance interconnection network. The MIMD programming model is the most appropriate to be used with the multicomputer systems. The multicomputers are also called **MIMD message-passing computers** [24].

Examples of Multicomputer Systems

Multicomputers can vary widely in terms of performance, cost or architecture.

Massively Parallel Processors (MPPs) are huge and expensive computers, consisting of possibly thousands of CPUs [25]. The CPU types used in a MPP machine are the CPU types commonly present in PCs or workstations. On the other hand, the CPUs are connected by a high-performance proprietary network, designed to achieve low latency and high bandwidth. The structure of the interconnecting network of the MPP systems normally employs hypercube, tree, or 2-D/3-D mesh topologies [26, 25].

Clusters of Workstations (COWs) are a popular form of multicomputers. They are basically a collection of standard PCs or workstation, connected through off-the-shelf communication components [25]. This approach makes the COWs simple to build and cheap compared to MPPs. COWs are sometimes also called **Network of Workstations**, defined as a computer network, which connects several computer workstations together and which can be used as a single cluster by utilizing specific software.

The cluster model is characterized by G.F. Pfister [27] as follows:

Definition 2.1. *A* **cluster** *is a type of parallel or distributed system that (1) consists of a collection of interconnected computers and (2) is utilized as a single unified computing resource.*

The element that makes the difference between the COWs and MPPs systems is the interconnection network. In contrast to the MPPs, the clusters' interconnection network is normally based on commodity LAN (Local Area Network) technologies. Besides that, the cluster nodes have a complete and conventional operating system.

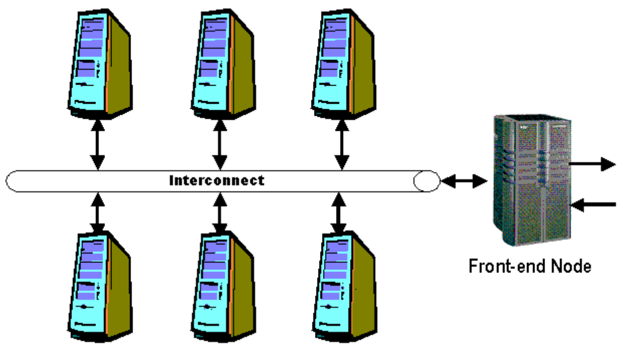


Fig. 2.4. Model of a cluster

The cluster model (Fig. 2.4) usually provides a front-end node, which represents the access point to the rest of the cluster's nodes, and additional software to provide a unified virtual view of the whole system as a single parallel computer. Within the specific software collection, an important role is played by the resource manager, which runs onto the front-end and allows the users to run their applications onto the cluster's computing nodes.

SMP clusters combine the performance of multiprocessor system with the scalability of the multicomputer systems. The nodes of the SMP cluster employ the SMP architecture.

Homogeneous versus Heterogeneous Multicomputers

Distributed computer systems are further classified in homogeneous and heterogeneous computer systems. This distinction applies most commonly to the multicomputer systems.

In **homogeneous computer systems**, the interconnection network uses the same technology everywhere, respectively all the CPUs have the same architecture and have access to memory of identical sizes and types. The homogeneous multicomputers tend to be used in the same way like multiprocessors, working on a single problem.

The heterogeneous computer systems comprise collections of distinct computers, which may widely vary in terms of processor type, memory size, and performance. Not only the CPUs, but also the interconnection network, or the software present onto the comprising computers may vary. There are drawbacks of using heterogeneous systems like high communication overheads, or compatibility issues, due to the variety of the hardware, operating systems or the software used, but these systems have the advantage of using already existing hardware and can provide a very competitive cost to performance ratio.

2.1.4 The Grid

Grid computing [10] represents the next evolution step following the cluster computing. Recently, Grid computing has faced rapid advances, widespread deployment and considerable hype. Grid computing offers a model for solving massive computational problems by making use of the unused resources (e.g. CPU cycles, disk storage) of large numbers of disparate, often desktop, computers.

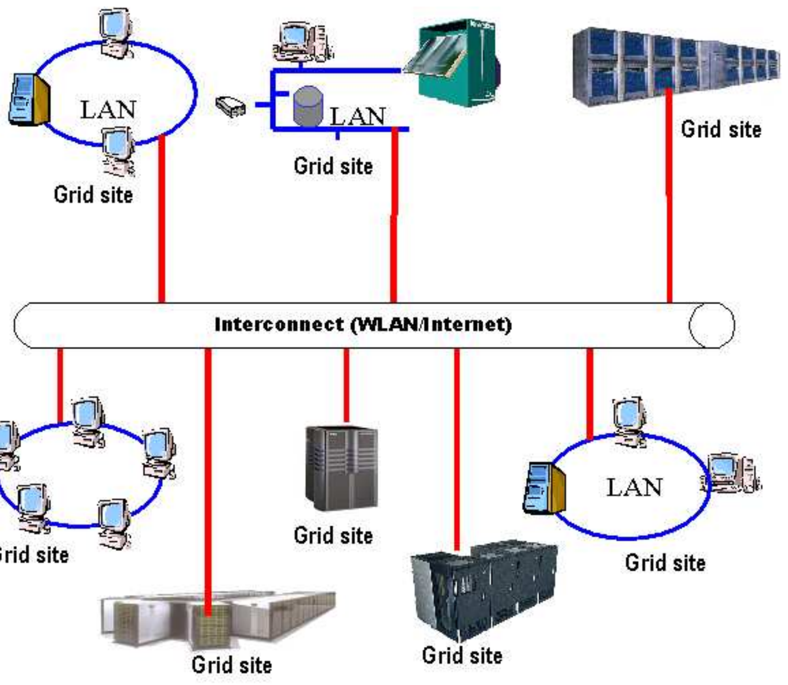


Fig. 2.5. Grid environment

Grid computing is about pooling and coordinated use of large sets of distributed resources. The resources may be computers, storage space, software and data, all connected to the Internet and a software layer that provides basic services for security, monitoring, resource manager, discovery, etc.

Dedicated clusters are characterized by close proximity of the comprising nodes, and they normally employ homogeneous hardware. In contrast, grids are inherently heterogeneous (in terms of hardware, interconnection network, existing software and operating system, etc.), and are characterized by a distant proximity of the (Grid) sites.

Initially, the term Grid was used to denote a computational Grid, defined as:

Definition 2.2. A computational Grid *is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities, across multiple administrative domains [8].*

The Grid computing is commonly defined through an analogy with a power grid [8, 9], which provides standard, reliable and low-cost access to common electric power, whilst it hides its actual source. Similarly, a Grid infrastructure consists of diverse resources, such as computers (e.g. workstations, PCs, clusters, MPPs or multiprocessors), networks, or storage space (Fig. 2.5), which authorized users can access. However, the use of individual resources will not be visible to the users and the users may not be aware of how the resources are assembled. Grid computing offers standards that enable completely heterogeneous systems to work together to form the image of a large virtual computing system. The users of the Grid can be organized dynamically into a number of **virtual organizations**, each with different policy requirements, which can share their resources collectively.

The Grid environment comprises a set of Grid sites (Fig. 2.5), each of them representing a distinct administrative domain.

Definition 2.3. *A* **Grid site** *represents the aggregation of Grid services within a single organization. It consists of a set of computational resources managed by one single hosting environment and one single resource manager (e.g. GRAM - Globus Resource Allocation Manager service [28]).*

A Grid site consists of a set of **computational nodes**, which can be any computing platform, ranging from single processor workstations or PCs, to SMP machines or MPP systems. Normally, a computational node is identified by its unique IP address or host name. The computational nodes can communicate with each other via a local network, which commonly provides high bandwidth, whilst Grid sites exchange data via a wide area network with lower bandwidth and higher latency. In many cases, depending on particular configuration or security settings (e.g. private IP addresses, firewalls), computational nodes from two distinct Grid sites cannot communicate directly.

2.2 Programming Model

In this section, we investigate the applications that can use the above-described computing architectures. We start with applications that are designed to run in single address space. Then, we analyze several types of distributed applications. Among these, workflow applications will play an important role further in the thesis (Chapter 5). We further outline the characteristics of the applications that use the Grid infrastructure, as a particular type of distributed applications.

2.2.1 Single Address Space Applications

In order to describe the single address space applications, we first define the notion of the address space:

Definition 2.4. *The* **address space** *is the range of memory locations that a process or processor can access. Depending on context, this could refer to either physical or virtual memory [29].*

Definition 2.5. *A* **single address space application** *is an application that is designed to run in a single address space.*

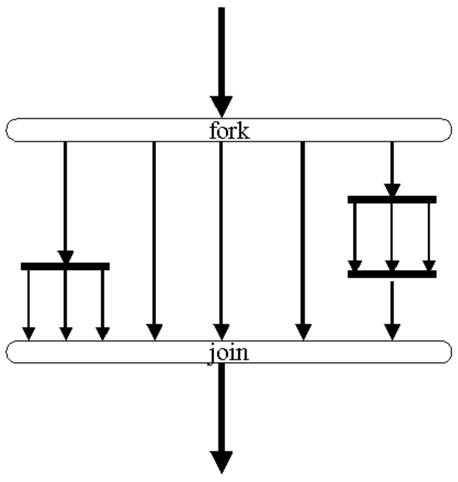


Fig. 2.6. Multithreading for single address space applications.

A single address space application consists of a set of instructions that manage data placed into the local memory of the system, no matter if the memory is physical or virtual. To run such an application, von Neumann machines with a single CPUs, or shared-memory machines such as SMP nodes can be used. In a distributed context, these are identified by their IP address, as a unique access point. In Grid context, such a machine corresponds to one computational node, which may be part of a Grid site.

Single address space applications can support parallelism as well. We call it **inter-process parallelism** if two or more application instances (commonly called processes) are allowed to run in parallel and compete for the resources of the machine (e.g. CPU, memory, I/O). The inter-process parallelism is also called **multitasking** and it is usually managed by the operating system, which supports **scheduling** (i.e. deciding which task may be the one running at a given time) and **context switching** (i.e. the act of reassigning a CPU from one task to another).

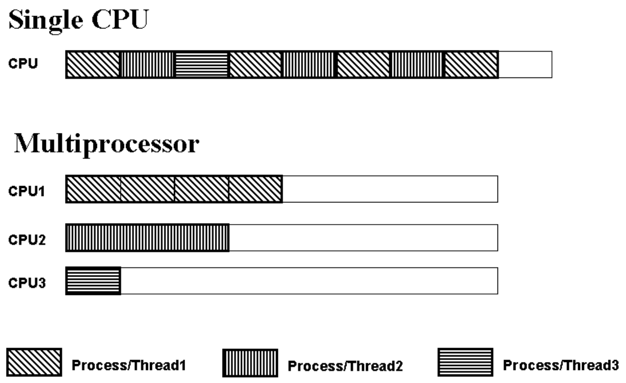


Fig. 2.7. Multiple threads/processes. Multiprocessor vs. von Neumann machine model

In case of **intra-process parallelism**, also named **multithreading**, an application spawns multiple threads (Fig. 2.6), which share the same memory context and process state information. Both multitasking and multithreading can benefit from the multiprocessor architecture (see Fig. 2.7).

2.2.2 Distributed applications

In contrast to single address space applications, **distributed application** are designed to use multiple computing resources (Section 2.1.3), physically placed in distinct locations and individually identified by their IP addresses or host names. The key issue is that the distributed resources do not share a common physical memory and the communication among them is significantly more expensive. It is possible to implement a **distributed shared memory system**, which mimics the functionality of the shared memory, by assembling the distributed memories of these resources, but this solution requires a *coherence protocol* in order to maintain the memory consistency according to a consistency model, which increases the complexity and affects the system scalability.

In the following, we investigate several types of distributed applications.

Collaborative distributed applications

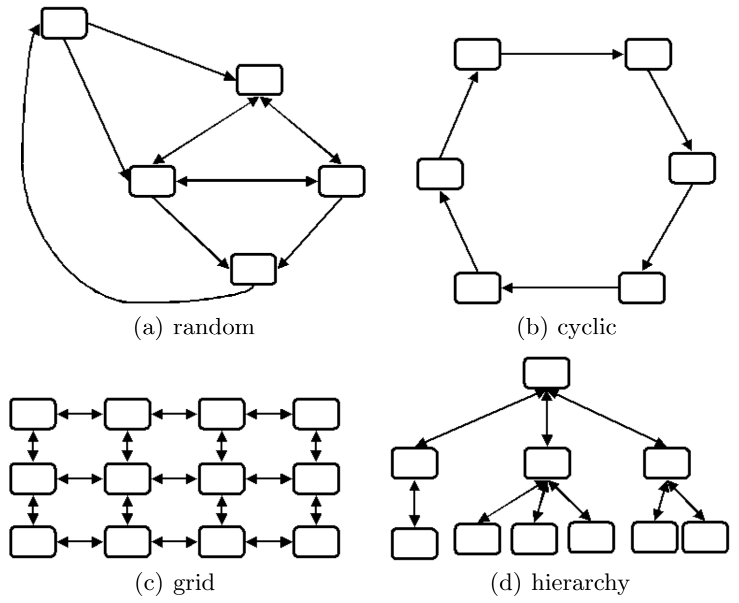


Fig. 2.8. Collaborative applications. Various communication patterns

Collaborative distributed applications consist of several components (commonly named (sub)jobs, tasks or activities), which may be freely placed onto any available resources (Fig. 2.8). Each component alternates computation with communication. The components may freely exchange data at random times by using available communication mechanisms (e.g. MPI [15], RPC [16], RMI [20], SOAP [30]).

Since no restriction applies to the interactions between the application components, it is not possible to apply a generic strategy of mapping them onto resources so that the overall performance is optimized. Each application may employ itself a specific optimization strategy.

Collaborative distributed applications are the most generic type of distributed applications. Applying restrictions to the application structure and/or communication among the components, we get particular types of distributed applications such as meta-tasks or workflows, which will be discussed in the following.

Meta-tasks

The meta-tasks consist of multiple independent components, commonly called *tasks*. There is no communication or synchronization between these tasks, and there is no constraint regarding the temporal order in which the tasks may run. Meta-tasks commonly solve **embarrassingly parallel problems** and represent a popular category of applications in distributed computing.

Definition 2.6. *An* **embarrassingly parallel problem** *is a problem for which no particular effort is needed to segment it into a very large number of parallel tasks, and there is no essential dependency (or communication) between the parallel tasks. [31]*

A popular example of meta-tasks is the **parameter sweeps**, also called **parameter studies applications**. In this case, the same application (commonly a single address space application) is executed on a large number of distinct input parameters/data sets. The multiple instances of the application are independently executed onto a large set of computing resources.

Meta-tasks are managed by resource management software, which determines the computing resource that should be used for each task, and which starts the task when the resource is available.

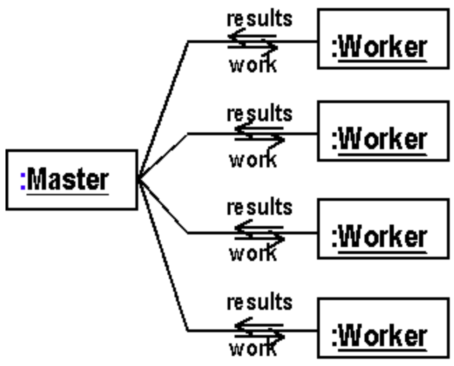


Fig. 2.9. Master/Worker programming model

The **master/worker** (called sometime also master/slave) programming model (Fig. 2.9) is similar to meta-tasks. In this model, a *master* program spawns several copies of the *worker* program (or uses already existing instances of this), and delegates them work/work items. There is no communication between the workers. However, the master sends input data to the workers and collects back results. In this case, the master plays the role of the resource management system explained above.

Workflow Applications

For workflow applications, the dependencies between the application components obey certain constraints. Meta-tasks can be seen as a particular case of workflows, which lack dependencies between the components. In the following, the workflow model is explained in detail.

Started as an initiative of the business community, the Workflow Management Coalition (WfMC) [22] aims to produce common terminology and standards for the exploitation of the workflow technology. According to WfMC:

Definition 2.7. A workflow *is the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules. [32]*

A business process represents a set of linked procedures and activities that realise a common goal. Furthermore, a **process definition** is used to define the automation of a business process, and a **workflow management system (WfMS)** provides software components to manage the workflows.

The basic element of the workflow is **the activity**. This is defined as:

Definition 2.8. An activity *is a description of a piece of work that forms one logical step within a process. [32]*

The activity is considered the smallest unit of work, which is scheduled by a workflow engine during workflow process enactment. The activity may be categorized as **automatic** activities (i.e. which can be managed automatically by a WfMS) or **manual** activities (which typically are not capable of automation and are managed by humans). In this thesis, we deal only with automatic activities, which will be simply called **(workflow) activities**. Moreover, we assume that an activity uses only a single computing resource at a time.

The so-called **dummy activities** represent a special type of activities:

Definition 2.9. A dummy activity *is an activity which has no inherent processing related to the business process, but which is used to represent and evaluate complex routing or process control conditions which may be too complicated to define efficiently using conventional process definition notation. [32]*

Since dummy activities do not require computational effort, they may be manipulated distinctively by the WfMS scheduler.

For each workflow, the WfMS produces a **process instance** (i.e. a single enactment of the process), respectively **activity instances** (i.e. single enactment of the activity) within the process instance. The WfMS manages also **the transitions** between the activities.

Definition 2.10. *A* **transition** *is defined as a point during the execution of a process instance where one activity (instance) completes and the thread of* *control passes to another, which starts. A transition may be associated with a* **transition condition***, which is a logical expression which may be evaluated by a workflow engine to decide the sequence of activity execution within a process. [32]*

The transitions between the activity instances are enabled in accordance with **dependencies** between activities, defined within the process definition. We identify two types of activity dependencies: **data-dependency**, which implies transitions that require data transfer (e.g. files, messages) between activities; **control-dependency**, which implies transitions that require only synchronization between activity instances and do not require data transfer. In both cases, the definition of the transition implies that the first activity instance (called **transition source**) has to finish, before the second one (called **transition target**) starts.

The data that is transferred between activity instances is called **application data**.

Definition 2.11. Application data *is application-specific data that is not accessible to the WfMS. In contrast,* **workflow control data** *is data that is managed by the WfMS, is internal to WfMS and is not normally accessible to applications. A hybrid type of data is represented by the workflow relevant data, which may be manipulated by both the workflow applications and the WfMS, in order to dynamically influence the transitions between the activity instances.*

Commonly, a graphical representation is used as a formalized view of the business process. Typically, this is a directed graph, which we call **workflow graph**. The workflow activities/activity instances are represented as the vertices of the workflow graph, whilst the transitions correspond to the edges of the graph. Since we consider two types of activity dependencies, we distinguish two types of edges in the workflow graph, as well: **Data-edges**, which are associated with data-dependencies, and **control-edges**, which are associated with control dependencies.

Most of the related work assumes that the workflows are modelled as Directed Acyclic Graph (DAG) workflow, in which case the associated workflow graph has no cycle. In our framework, several elements are added to this model.

- A **conditional branch** is a point in the workflow execution which indicates that transition conditions need to be evaluated. The conditional branch is not associated with computation and can be represented as a vertex in the workflow graph. All outgoing transitions for a branch need to be associated with transition conditions.
- An **initial state** represents the start of a workflow execution. It is not associated with computation and can be represented as a vertex in the workflow graph.

- A **final state** represents the end of a workflow execution. It is not associated with computation and can be represented as a vertex in the workflow graph.
- A **(sequential) loop** models the repetitive execution of one or more workflow activities for a previously determined number of times or until a termination condition is met. The loop is represented as an edge (of a distinct type) in the workflow graph.
- A **parallel loop** models the execution of n identical copies of one or more workflow activities. The parallel loop is represented as an edge in the workflow graph.

The workflow model which is used in this thesis is further detailed in Section 5.1.

2.2.3 Grid Applications

What we usually call a Grid application (or Grid-enabled application) is basically an application, which can run on the Grid, and uses for this purpose the Grid software infrastructure. Bart Jacob et al. [2] define a Grid application in term of its constituent jobs:

Definition 2.12. *A* **Grid Application** *is a collection of work items (jobs/tasks) designed to solve a certain problem or to achieve desired results using a Grid infrastructure. In other words, a Grid application may consist of a number of jobs that together fulfil the whole task.*

The job *is considered as a single unit of work within a Grid application. It is typically submitted for execution on the Grid, has defined input and output data, and execution requirements in order to complete its task. A single job can launch one or several processes on a specified node. It can perform complex calculations on large amounts of data or might be relatively simple in nature [2].*

In short, a Grid application is a collection of jobs that carry out a complex computing task by using Grid resources.

Currently, the Grid follows the service-oriented approach [11]. The Open Grid Services Infrastructure (OGSI) provides specifications for query, monitoring, discovery, factory, notification, security, registration, management, scheduling, and other functions that can be made available to all Grid users. A Grid application can use these registered services, along with Grid infrastructure (Fig. 2.10), to accomplish specific work-related tasks that solve business and technical problems.

On the other hand, Bart Jacob et al. [2] have identified three types of Grid applications, in terms of *application flow*, defined as the flow of work between the jobs that make up the Grid application:

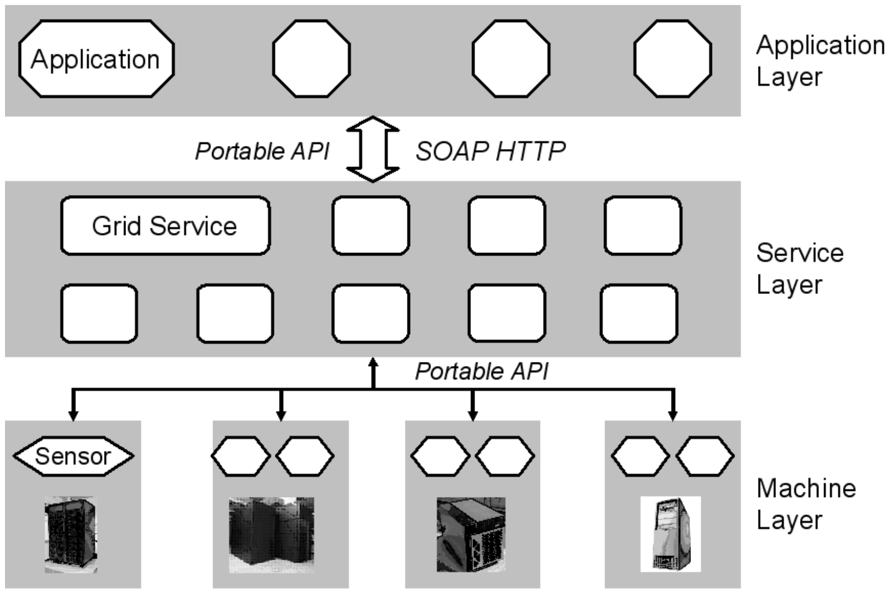


Fig. 2.10. Grid service-oriented model

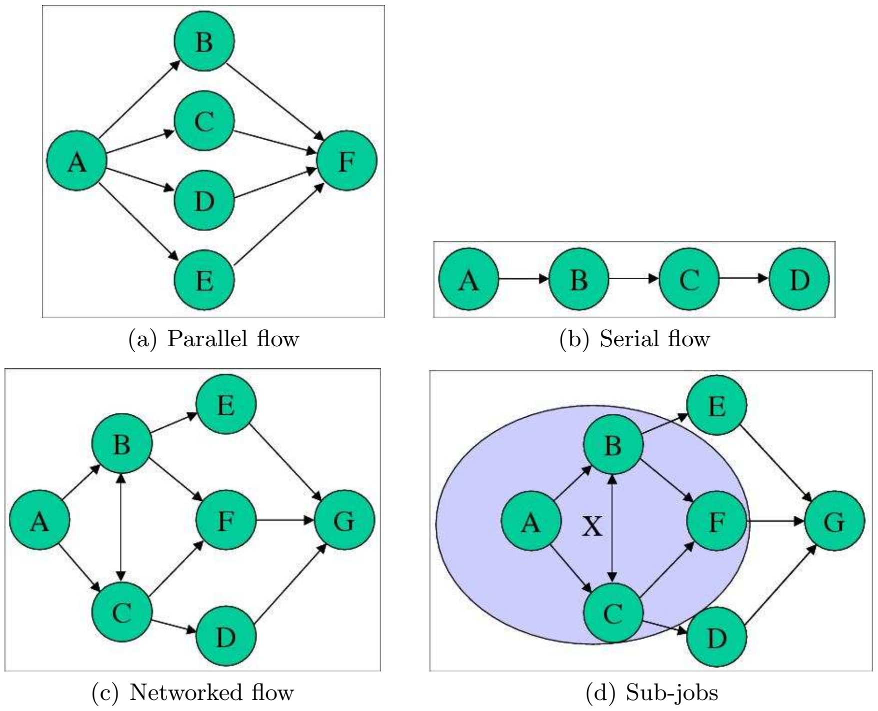


Fig. 2.11. Application flow types in Grid applications [2].

- **Parallel flow applications** (Fig. 2.11(a)) are similar to meta-tasks. The jobs of the application can be executed in parallel, and there is no (or a very limited) exchange of data among the jobs. An initial job may be used to launch a number of jobs on preselected or dynamically assigned Grid nodes. Each job may receive a set of data, perform independent computation and deliver its output. A final job may be used to collect the output data from all these. Grid services, such as a resource broker, scheduler and/or enactment engine, may be used to determine best suitable resource, the appropriate time frame for execution, and to launch the execution of each job.
- **Serial flow applications** (Fig. 2.11(b)) resemble workflows with a limited structure (e.g. only the sequence of activities is allowed). Such an application has a single thread of job execution, and each job produces data that is used by the subsequent job as input. The advantages of running in a Grid environment are not based on access to multiple resources in parallel, but rather on the ability to use one of the several appropriate and available resources, especially if particular jobs require specialized resources.
- **Networked flow applications** (Fig. 2.11(c)) are similar to distributed collaborative applications or complex workflows. In this case, a job flow management service is required to handle the synchronization of the individual results. Loose coupling between the jobs avoids high inter-process communication and reduces overhead in the Grid. The complexity of such an application adds more dependencies on the Grid infrastructure services such as schedulers and brokers, but once that infrastructure is in place, the application can benefit from the flexibility and utilization of the virtualized computing environment.

  In many cases, a subset of jobs may be seen as the sub-jobs of a larger job (Fig. 2.11(d)), in a hierarchical system. The reason for using such a system is that the higher-level jobs could include the logic to obtain resources and launch sub-jobs in the most optimal way. In this way, some very large applications may get benefit from passing the management of certain tasks to the individual components.

We observe that the Grid applications are quite similar to the distributed applications discussed in the previous sections. The main difference between the two classes is the ability of the Grid applications to use the Grid infrastructure (e.g. physical resources and software services) in order to obtain performance from large amount of distributed computation power. However, not all applications can be modified to run on a Grid and achieve scalability, and there are no tools for automatic transformation of the existing applications into grid-enabled applications.

David Kra [33] identifies 6 strategies to build a grid-enable application:

- **Batch Anywhere.** This strategy has the goal of running one application instance on one of the available Grid nodes. We obtain what is commonly called a **single-site Grid application**, which uses a single Grid site to run (see Section 2.1.4). This can be a single-address space application (Section 2.2.1) or a distributed application managed by a single local resource manager.

- **Independent Concurrent Batch.** This strategy supports multiple independent instances of the same application running concurrently, potentially using distinct input data, similar to a meta-task.
- **Parallel Batch.** This strategy implies the existence of a client that splits work into multiple server jobs and assembles intermediate results into a final output (similar to the master/worker model). The server jobs behave as the multiple independent tasks in a meta-task.
- **Service.** The service is software, which is already placed onto a Grid node, and commonly started before its first use. In this case, the programming efforts switch to service implementation, in contrast to the previous strategies. Clients access the service only by using Grid middleware, and this invokes the service on client's behalf. The service may be shared among independent clients, and it can maintain its state between calls.
- **Parallel services** combine services with parallel batch strategies, by providing multiple service instances that can be invoked in parallel (independently) on client's behalf.
- **Tightly Coupled Parallel Programs** imply intense communication and synchronization between clients and services, respectively among services (similar to the workflow or collaborative distributed applications). This sort of applications commonly requires significant programming effort and complex cooperation among the services. Applications of this type are usually specialized applications that comprise extensive computation and would take even decades if running on single machines.

3

JavaSymphony Programming Paradigm

There are numerous research efforts to provide flexible and high-level APIs that support programming of parallel and distributed applications. Much of this work focuses on providing automatic performance control (e.g. automatic load balancing, mapping, migration, or control of locality) for distributed applications and assumes that the runtime system is able to detect parallelism, exploit locality and achieve efficient load balancing. However, fully automatic systems commonly cause poor performance results, due to the lack of information about the application and insufficient static and dynamic analysis.

On the other hand, many programmers are well aware of how to structure a distributed application, where to place objects, which objects interact with each other, and how to exploit and to control locality and parallelism. JavaSymphony, on the one hand, supports automatic mapping, load balancing, and migration of objects without involving the programmer. On the other hand, in order to enhance the performance of distributed applications, JavaSymphony provides a semi-automatic mode, which leaves the error-prone, tedious, and time consuming low-level details (e.g. creating and handling of remote proxies for Java/RMI) to the underlying system, whereas the programmer controls the most important strategic decisions at a very high level.

This chapter presents in detail the JavaSymphony programming paradigm. The JavaSymphony programming API has two key components: dynamic virtual distributed architectures (VAs), which we discuss in Section 3.2, respectively JavaSymphony objects (JS objects), presented in Section 3.3. We further demonstrate additional useful programming elements, including: a variety of remote method invocation types (synchronous, asynchronous and one-sided method invocation); (un)lock mechanism for VAs and JS objects; high level API to access a large variety of static or dynamic system parameters, selective remote class-loading; automatic and user-controlled mapping of objects; conversion from Java conventional objects to JS objects for remote access; single-threaded versus multi-threaded JS objects; object migration; distributed event mechanism; synchronization mechanisms.

3.1 JavaSymphony Applications

The developers may use the JavaSymphony programming paradigm to easily build Java-based distributed applications, which we call JavaSymphony applications (JS applications). Commonly, every JS application must first register with the JavaSymphony Runtime System (JRS) (detailed in Chapter 4). This is realized by using the *register* method of the *JSRegistry* class. Thereafter, in order to manage remote distributed computing resources, VAs can be requested. In order to minimize performance problems due to Java class loading, all required classes are stored in Java archive files and loaded onto arbitrary nodes of a defined VA. Objects can be created, mapped, and migrated both on a local, as well as on a remote computing node. JavaSymphony supports three types of (remote) method invocations, namely synchronous, asynchronous, and one-sided method invocations, which enable the application's objects to collaborate according to the application logic. In the end, an application should un-register from the JavaSymphony Runtime System, by using the *unregister* method of the *JSRegistry* class.

The JS application is linked with the JavaSymphony class library and is executed in the same way as any regular Java application. The programming elements are further discussed in detail in the next sections.

3.2 Dynamic Virtual Architectures

JavaSymphony introduces the concept of *dynamic virtual distributed architectures* (called VAs), which allows the programmer to define a structure of a heterogeneous (e.g. type, speed, or configuration) network of computing resources and to control the code placement and the mapping, load balancing, and migration of objects.

Dynamic virtual distributed architectures (see Fig. 3.1) consist of a set of components, each of which associated with a level:

- level-1 VA corresponds to a single computing node such as a PC, workstation or a multiprocessor system.
- level-2 VA refers to a cluster of level-1 VAs (e.g. workstation or PC cluster).
- level-3 VA defines a cluster of geographically distributed level-2 VAs connected, for instance, by a wide area network.
- level-i VA with $i \geq 2$ denotes a cluster of level-($i-1$) VAs, which, among the others, allows to define arbitrary complex heterogeneous GRID architecture distributed across several continents.

In the following sections, we refer the level-1 VAs as (computing) nodes.

3.2.1 System Constraints

A key advantage of JavaSymphony over other systems is the provision of a high-level API to static and dynamic system parameters. Based on the system

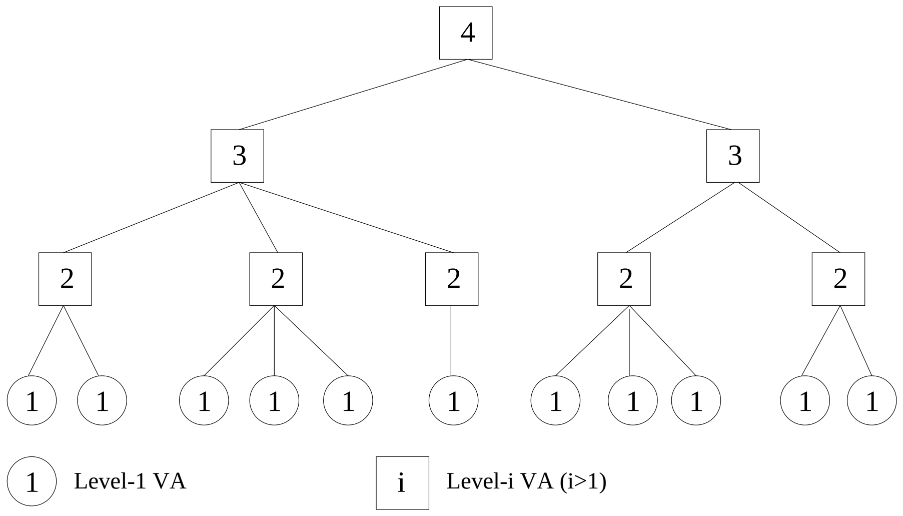


Fig. 3.1. Example of a JavaSymphony level-4 virtual architecture

parameters, the user can specify system constraints to control load balancing, migration, mapping, to honour computing site policies, etc. The basic idea is to include architecture components in a VA that obey user-defined constraints defined over system parameters. On the other hand, the constraints can be used to examine the properties of physical resources, which include static parameters such as machine name, operating system, peak performance parameters, etc., or dynamic parameters such as system load, idle times, and available memory. During the execution of an application, the static parameters remain unchanged, while the dynamic parameters may change. A list of system parameters that are supported by JavaSymphony is presented in Appendix 10.2.

Constraints are added to a *JSConstraints* object by invoking calls to the following method:

setConstraints(sys_param,rel_op,[float_val|int_val|string_val]);

```
JSConstraints constr = new JSConstraints();
constr.setConstraints(JSConstraintsConst.C_HOST_URL,"!=","r2d2");
constr.setConstraints(JSConstraintsConst.C_CPU_IDLE,">=",90.0f);
constr.setConstraints(JSConstraintsConst.C_MEMORY_FREE_KB,">=",10240);
```

Fig. 3.2. Building constraints. Code excerpt

Each method invocation adds a constraint with the following pattern:

sys_param_name rel_op value

where *rel_op* corresponds to arbitrary relational operators and *value* refers to floating point/integer numbers or strings. For instance, consider the JavaSymphony code excerpt shown in Fig. 3.2. A set of constraints is collected in the *constr* object. These specify that the VA's computing nodes may not include the one whose name is "r2d2", their CPU should be idle for more than 90%, and they should have at least 10240 Kbytes of unused memory.

```
    //* get system parameters for level-1 VA v1: CPU idle time, URL, swap space
float cpuIdle = v1.getSysParamAsFloat(JSConstraintsConst.C_CPU_IDLE);
String sURL = v1.getSysParamAsString(JSConstraintsConst.C_HOST_URL);
int swap = v1.getSysParamAsInt(JSConstraintsConst.C_SWAP_SPACE_AVAIL);
```

Fig. 3.3. Retrieving system parameters. Code excerpt

The programmer can define constraints over a large number (depending on the operating system and the installed system sensors) of various system parameters. JavaSymphony provides easy access to VA's system parameters, as illustrated in the Fig. 3.3.

3.2.2 Creating VAs

```
JSConstraints constr;
    //* request level-1 VA
VA v1 = new VA(1);
    //* request level-1 VA for which constraints hold
VA v2 = new VA(1, constr);
    //* bottom-up request for level-2 VA by adding existing VAs to it
VA v3 = new VA(2);
v3.addVA(v1);
v3.addVA(v2);
    //* top-down request for level-4 VA (see Fig.  3.1) with 2 level-3 VA's:
    //* first level-3 VA with 3 level-2 VAs with 2, 3,
    //*           and 1 level-1 VAs, respectively
    //* second level-3 VA with 2 level-2 VAs with 3
    //*           and 2 level-1 VAs, respectively
VA v4 = new VA(4, new int[][] {{2,3,1}, {3,2}});
```

Fig. 3.4. VA creation. Code excerpt

The JavaSymphony class *VA* is used to define the topology for VAs. As we have mentioned before, a node (level-1 VA) corresponds to a single computing resource (e.g. PC or workstation), whereas VAs with higher levels denote collections of lower level VAs. A set of constructors of the *VA* class may be used to create complex topologies with a single line of code. The task of allocating and managing VAs is left to the JRS. Moreover, VAs can also be built in a bottom-up manner, by assembling lower level VAs into one higher level VA (Fig. 3.4).

3.2.3 Modifying VAs

The JRS returns a handle for every generated VA. These handles are first order objects that can be passed as parameters to methods. Any thread with a handle to a VA has access to and can modify or even release this VA. A lock/unlock mechanism provided by JavaSymphony can prevent concurrent changes to VAs. If a thread t has a handle to a VA v and locks v, then no other thread can access v until thread t unlocks v again. A lock operation on a VA v is delayed until all unfinished methods on v have completed their execution. It is recommended to use the lock/unlock mechanism in order to avoid inconsistent modifications of the VAs.

Figure 3.5 illustrates the modifications applied to *v1* inside the lock/unlock code region, as shown in the code excerpt from Fig. 3.6.

3.2.4 Retrieving Information about VA's

Each VA may be queried for specific information, such as system parameters (static or dynamic) or information about the topology (e.g. the parent or the successors in the tree structure of a VA). The *getLocalNode* static method returns the local computing node (i.e. the level-1 VA corresponding to the machine onto which the current execution thread is running). The code excerpt in Fig. 3.7 demonstrates these features.

System constraints can be used to control load balancing, to improve the program's performance, to honour computing site policies, etc. The program can check if the constraints that have been used at the creation of the VA, or any other constrains, (still) hold or not for a specific VA, as demonstrated by the code excerpt in Fig. 3.8.

3.2.5 Class Loading

JavaSymphony enables the programmer to generate objects both locally and remotely. As JavaSymphony is built on top of the Java RMI mechanism, it is required that all objects that can be created remotely to be serializable (i.e. implements *java.lang.Serializable* interface). Before an object can be generated, the class file of this object must be located either locally in the CLASSPATH or at an accessible URL. JavaSymphony assumes that all Java class files are available at a given VA before objects are generated. This reduces

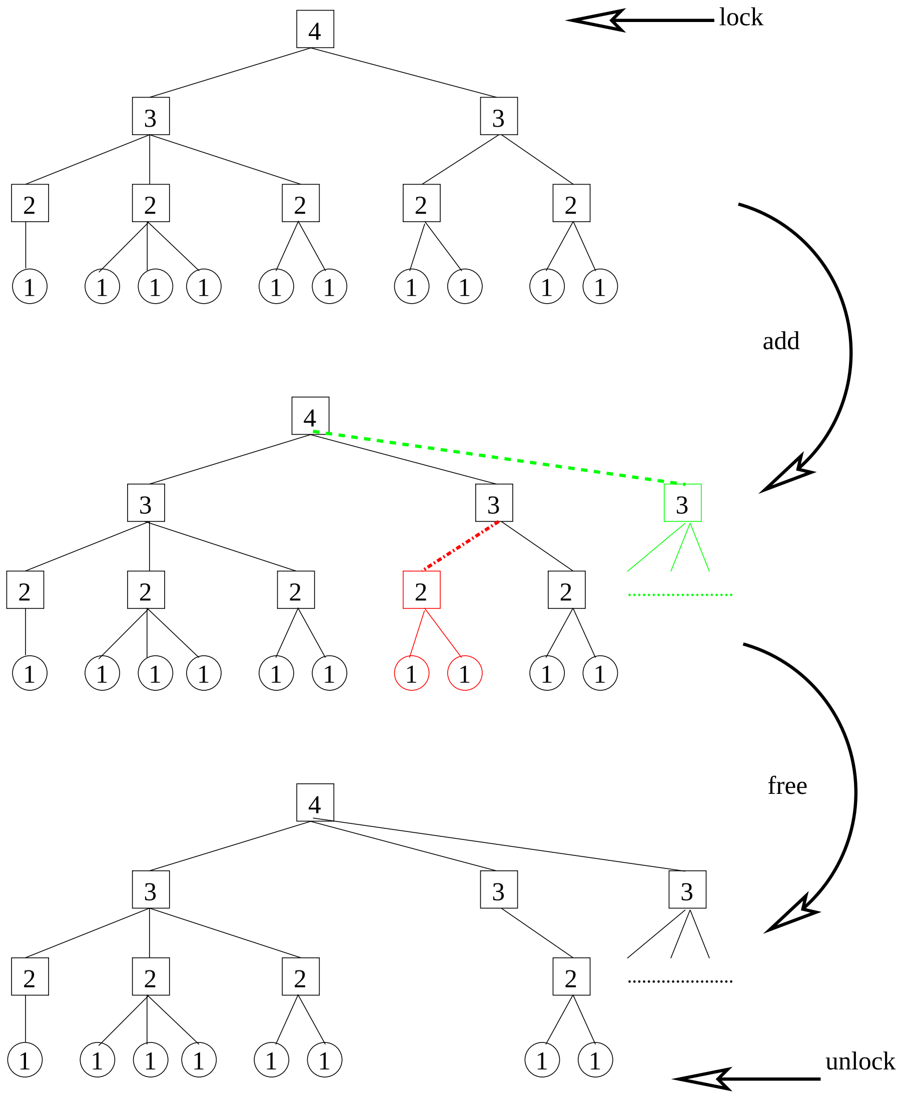


Fig. 3.5. Dynamically changing VAs by using the (un)lock mechanism

```
VA v1 = new VA(4, new int[][] {{1,3,2}, {2,2}});
VA v2 = new VA(3, new int[] {4,2,1});
...
v1.lock();          //* lock v1 before modifying it
v1.addVA(v2);       //* change v1
v1.free(new int[]{2,1}) //* delete the first successor of the second successor of v1
v1.unlock();        //* unlock v1
v1.free();          //* delete v1 when it is not needed anymore
```

Fig. 3.6. VA modification. Code excerpt

```
   //* get system parameters for v1 - a level-i VA
float cpuIdle = v1.getSysParamAsFloat(JSConstraintsConst.C_CPU_IDLE);
String sURL = v1.getSysParamAsString(JSConstraintsConst.C_HOST_URL);
int swap = v1.getSysParamAsInt(JSConstraintsConst.C_SWAP_SPACE_AVAIL);
   //* obtain the VA level and parent VA of v1
VA v2 = v1.getPred(v1.getLevel()+1);
   //* obtain the n-th successor of v1
VA vn = v1.getVA(n);
   //* obtain the local level-1 VA
VA v_local = VA.getLocalNode();
   //* obtain the 2nd node of the 2nd level-2 VA
   // *** of the 1st level-3 VA of a level-4 VA
VA v4 = new VA(4, new int[][] {{1,3,2}, {2,2}});
VA v1 = v4.getVA(new int[]{1,2,2});

   //* number of level-i VAs in v4
int countLevel_i = v4.nrVA(i);
   //* traverse level-1 VAs of v4
VAEnum e = v4.enumerateVA(1);
while (e.hasMoreVA())
{
   VA v5 = e.nextVA();
   ...
}
```

Fig. 3.7. Managing VAs' properties. Code excerpt

the amount of data transferred when creating objects. A Java archive file with all necessary byte code can be transferred during the application initialization phase, which saves multiple byte code transfers during the application execution. For this purpose, JavaSymphony facilitates the building of one or several codebases (i.e. collection of Java classes needed by an application to run), which can then be delivered as Java archive files to arbitrary components

```
JSConstraints constr;
      //* check if the initial constraints of a VA v still hold
boolean itHolds = v.constrHold();
      //* check if specific constraints hold for VA v
boolean itHolds = v.constrHold(constr);
```

Fig. 3.8. Checking VAs' constraints. Code excerpt

```
JSCodebase codebase = new JSCodebase(); //* initialize a codebase

   //* a Java archive or class file is added to the codebase
codebase.add("../classes.jar");
codebase.add("../testclasses.class");

   //* Java archive or class file is fetched from URL and added to the codebase
URL classURL = new URL("http://www.par.univie.ac.at/JS/test/file.class");
codebase.add(classURL);

codebase.load(v); //* load codebase to all nodes of a VA v
codebase.free(); //* free codebase
```

Fig. 3.9. Managing the codebase. Code excerpt

of a VA, by using the *JSCodebase.load* method. JavaSymphony, therefore, not only supports the programmer to control data (objects) locality, but allows him to control the code locality, as well.

The codebase management is demonstrated in Fig. 3.9. All or individual codebases can be dynamically transferred onto individual VAs. For instance, object migration commonly requires to transfer codebases in order to ensure that the class files for the objects to be migrated are available at the new location. By using the *free()* method on a codebase object, the codebase information is deleted, and associated memory is released.

3.3 JavaSymphony Distributed Objects

In order to use JavaSymphony to distribute regular Java objects onto virtual architectures, we need first to encapsulate them into so-called JS objects. Afterwards, they can be used by remotely invoking their methods. Moreover, the JS objects support features like migration and synchronization mechanisms that are highly useful for distributed computing.

3.3.1 Creation of JS Objects

Assuming that class files are available on every component of a VA where needed, JS objects can be created by generating instances of *JSObject* class. A set of *JSObject* constructors allows the developer to specify the Java class name of the object that is encapsulated in the JS object, the constructor arguments for this object, whether the JS object is single-threaded or multi-threaded, and the JS object location, with optional constraints. The provision of a level-1 VA indicates the exact location where the JS object will be placed. If a higher level VA v (with level greater or equal than 2) with/without constraints (see Section 3.2.1) is provided, then the JRS tries to determine a level-1 VA in v which honours all the given constraints. If only constraints and no location are provided, then the JRS searches for a location that fulfils all constraints, within the set of all available nodes. If neither location nor constraints are provided, then the JRS will use a default location based on configuration constraints (e.g. a level-1 VA with the smallest system load and reasonable resources available) set under the JavaSymphony Administration Shell (JS-Shell - see Section 4.2). The code excerpt in Fig. 3.10 illustrates the features described above.

```
VA v1 = new VA(1); //* allocate level-1 VA
VA v2 = new VA(4,....); //* allocate level-4 VA
VA vLocal = VA.getLocalNode(); //* get local level-1 VA
JSConstraints constr;
Object[] args = new Object[] { ... }; //* parameters for the new object

      //* generate an object of class "ClassName" at a VA decided by JRS,
      //* restricted by constraints or placed onto the local level-1 VA
JSObject obj1 = new JSObject("ClassName"[,args][,constr][,vLocal]);

      //* generate an object on a higher level VA;
      //* JRS decides onto which level-1 VA of v2 the object is generated
JSObject obj1 = new JSObject("ClassName",[args,] v2);

      //* generate an object onto a specific VA v1
JSObject obj1 = new JSObject("ClassName",[args,] v1);

      //* generate obj1 onto the level-1 VA where obj2 is placed
JSObject obj1 = new JSObject("ClassName" ,obj2.getVA());
```

Fig. 3.10. JS object creation. Code excerpt

Each JS object can be created as a single- or multi-threaded JS object (Fig. 3.11). This attribute of the object can be changed dynamically at run-time. A single-threaded object is associated with only one thread that executes all of its methods. In contrast, the JRS may assign multiple threads to a multi-threaded object can by the JRS that execute its methods simultaneously. Even a single method of a multi-threaded object can be executed by multiple threads in parallel. The number of threads incorporated to execute multi-threaded objects can be changed dynamically through the JS-Shell. A multi-threaded object can benefit from multiprocessor nodes (e.g. SMP nodes), since the several threads that execute in parallel a multi-threaded object's methods may be assigned on distinct processors. For a single-threaded object, since only one method at a time is allowed to run, inconsistent concurrent accesses to the object are prevented.

```
    //* generate a multi-threaded object in a node of
    //* a higher level VA v that honours a set of constraints
boolean multiThreaded = true;
JSObject obj1 = new JSObject(multiThreaded, "ClassName" [,args] [,constr][,v]);

    //* objects can be made single- or multi-threaded at runtime
obj1.singleThreaded();
obj1.multiThreaded();

    //* convert a conventional Java object to a JS object
ClassName obj = new ClassName(...);
JSObject obj2 = JSObject.convertToJSObject(obj [,multiThreaded]);
```

Fig. 3.11. Single- and multi- threaded JS objects. Code excerpt

The JS objects can be generated based on existing non-JS (conventional Java) objects, by using the *convertToJSObject* method. The first parameter of the method represents the non-JS object, and, optionally, the second parameter indicates whether a single- or multi-threaded object should be generated (Fig. 3.11). The migration (Section refsec:migrate-obj) of a JS object *obj2* that has been generated through conversion based on a non-JS object *obj* is possible; however, it must be carefully handled. In case of migrating *obj2* to another VA, all previous local references to *obj* should be deleted by the programmer to avoid programming errors and memory leaks.

3.3.2 Remote Method Invocation

In the current implementation of JavaSymphony, the communication among the remote distributed components of the JavaSymphony middleware is based on Java/RMI. Java/RMI imposes blocking remote method invocation, which prohibits overlapping of waiting time – for the results of the method invocations to arrive – with some useful local computations. In addition to synchronous (blocking) RMI, JavaSymphony also offers asynchronous (non-blocking) and one-sided (non-blocking without results) RMI.

All these three types of method invocations have similar signatures: A method name, a list of parameters and, optionally, a list of parameter types are provided as method parameters. The JRS uses the parameter types in order to identify the appropriate method to invoke, in case of an object that may have several methods with the same name and similar signatures. If the list of parameter types is given, the JRS will choose the method with the indicated signature.

Synchronous Method Invocation

A synchronous method invocation (see Fig. 3.12(a)) is initiated by calling the *sinvoke* method of *JSObject* class. The call of *sinvoke* blocks the calling site until the result arrives. The method's parameters are passed as an array of objects. JavaSymphony *sinvoke* always returns a result of class Object, which needs to be explicitly cast to the actual class of the result. Figure 3.13 demonstrates the usage of the synchronous method invocation in JavaSymphony. In the code excerpt, a method with name *methodName* is invoked for the *obj* JS object, with two parameters of type *Param1*, respectively *Param2*.

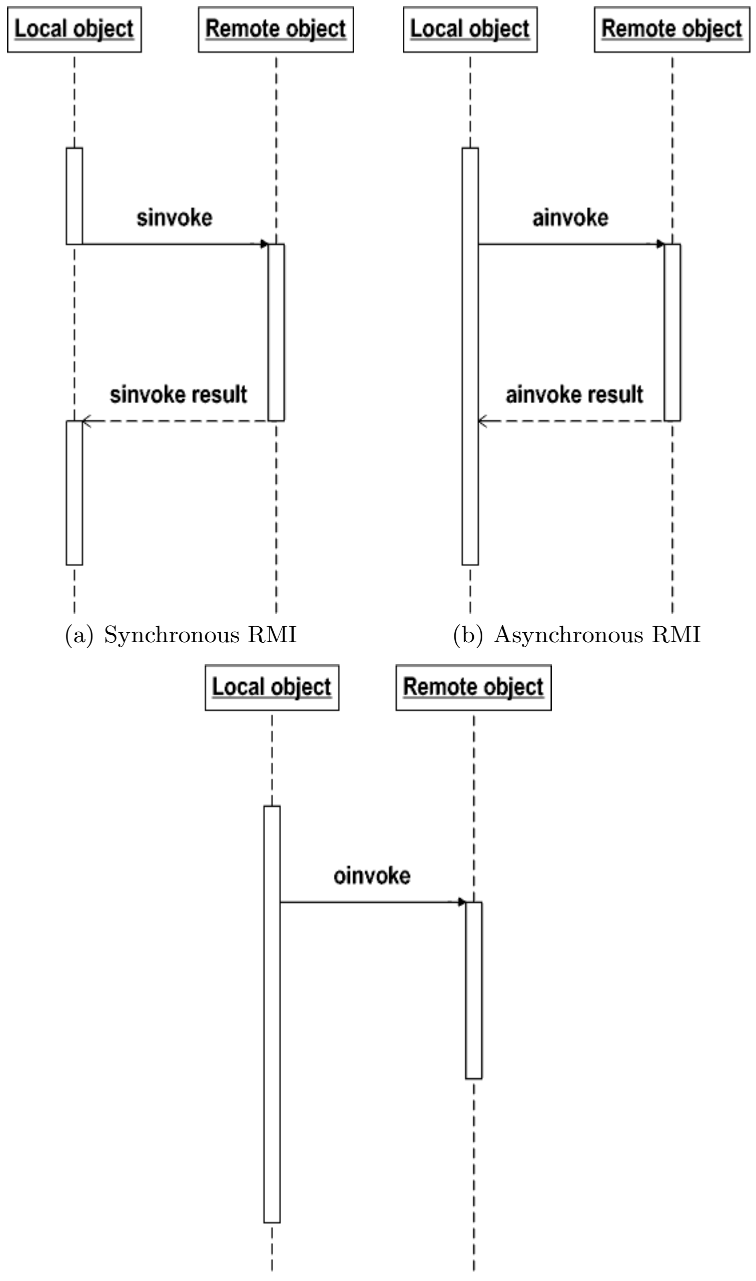
Asynchronous Method Invocation

Asynchronous method invocations (see Fig. 3.12(b)) are commonly employed to parallelize computations. An asynchronous method invocation is initiated by calling the *ainvoke* method of the *JSObject* class. Once again, an array of objects is used to hold the method parameters. The method call, however, does not block, but immediately returns a handle. At the calling site the execution continues, while the method is being executed. If a pre-defined method *handle.isReady* returns *true*, then the result is available, otherwise the method is still being executed. If the calling site wants to block until the result has arrived – for instance, because no other useful computations can be done – then the *handle.getResult* method may be called. Note that this method returns the result object of generic type *Object*, which needs to be explicitly cast to the actual class of the result. The code excerpt in Fig. 3.14 illustrates these aspects.

One-sided Method Invocation

An one-sided method invocation (see Fig. 3.12(c)) is initiated by calling the *oinvoke* of *JSObject* class.

One-sided method invocations are used when it is not necessary to wait for the completion of the invocation and the result is not required. The one-sided method invocation can improve the performance of the application because



(a) Synchronous RMI (b) Asynchronous RMI

(c) One-sided RMI

Fig. 3.12. Remote method invocation in JavaSymphony

```
JSObject obj = new JSObject("ClassName");
...
Object[] params = new Object[] {new Param1(), new Param2()};
Class[] paramTypes =
      new Class[] {Param1.getClass(), Param2.getClass()};
ResultClass result = (ResultClass)obj.sinvoke("methodName,"
      params [,paramTypes] );
```

Fig. 3.13. Synchronous RMI in JavaSymphony. Code excerpt

```
Object[] params = {new Param1(), new Param2()};
Class[] paramTypes = new Class[] {Param1.getClass(), Param2.getClass()};
      //* invoke remote method with parameters; a handle is returned
      //* to refer to the (future) method's result
ResultHandle handle = obj.ainvoke("methodName",
      params [,paramType]);
...
      //* verify whether result is available
if (handle.isReady()) {
      //* get result in blocking mode
      ResultClass result = (ResultClass)handle.getResult();
}
...
      //* wait for result to arrive in blocking mode
      //* without checking for the result to be available
ResultClass result = (ResultClass)handle.getResult();
```

Fig. 3.14. Asynchronous RMI in JavaSymphony. Code excerpt

```
Object[] params = {new Param1(), new Param2()};
Class[] paramTypes = new Class[] {Param1.getClass(), Param2.getClass()};
obj.oinvoke("methodName",params [,paramTypes] );
```

Fig. 3.15. One-sided RMI in JavaSymphony. Code excerpt

there is no need to transfer back a result from a node that hosts the remote object. Moreover, one-sided method invocation reduces some bookkeeping overhead of JRS. The usage of one-sided method invocations is demonstrated in the code excerpt of Fig. 3.15.

3.3.3 Migration of JS Objects

Objects can be migrated during the execution of the JS distributed application. The JRS, however, verifies before object migration, whether any of its methods are currently being executed. If so, then migration is delayed until all unfinished method invocations have been completed, otherwise the object can be immediately migrated. JavaSymphony offers two forms of object migration: automatic migration, which is controlled by the JRS, or explicit migration, which is controlled by the programmer. The programmer can also specify the destination VA, constraints, and whether the codebase(s) should be transferred. Optionally, specific codebase could be indicated, to be transferred before the migration.

Explicit migration can be encoded by the JS application programmer, based on system constraints. JavaSymphony allows access to the VAs' system parameters, as described in Section 3.2.1. In case of higher-level VAs, the system parameters for a level-i VA are the average values across its level-$(i-1)$ VAs, which is limited to system parameter values of numeric types (i.e. integer and float). The methods *getSysParamAsFloat*, *getSysParamAsInt* or *getSysParamAsString* can be used to examine the system parameters of interest. Moreover, it can be checked by using the *constrHold* method whether a set of constraints (see Section 3.2.1) currently hold for a given VA. For instance, in the code excerpt from Fig. 3.16, it is examined whether *v1*, the level-1 VA onto which the *obj* object resides, has less than 50 % idle time or whenever it does not fulfil a set of constraints. If so, the object can be migrated by using the *migrate* method of the *JSObject* class. If *migrate* is called without any parameters, the JRS decides where to send the object.

```
JSConstraints constr;
VA v1 = new VA(1); VA v2 = new VA(4,...);
JSCodebase cb; JSObject obj;
      ...
VA v3 = obj.getVA();
         //* check if v3 on which obj resides has less than 50 % idle time
         //* or constr do not hold for v3
if (v3.getSysParamAsFloat(JSConstraints.C_CPU_IDLE) < 50) ||
      !v3.constrHold(constr) )
{
      obj.migrate(); //* migrate object to a node destined by JRS
      obj.migrate(constr, true); //* migrate object to a node based on constraints
      obj.migrate(v1); //* migrate object to a specific node
      obj.migrate(v2); //* migrate object to a node of v2 to be destined by JRS

         //* migrate object and move codebase to the destination VA
      obj.migrate(va, cb);
      obj.migrate(constr, cb);
      obj.migrate(cb);
}
```

Fig. 3.16. Object migration in JavaSymphony. Code excerpt

The *migrate* method may use a level-1 VA as parameter, which indicates where to migrate the object. If constraints are provided, then the JRS finds a node that honours the constraints as the target node. A Java exception is thrown if no suitable node is found. The exception has to be handled by the application programmer.

3.3.4 Lock/Unlock JS Objects

The remote objects are accessed through handles (i.e. instances of *JSObject* class), which are first order objects. They can be passed to methods and, therefore, can be distributed onto VAs, as well. Any thread that has a handle to a JS object has access to it and can invoke its methods or even release this object. However, concurrent accesses to objects can be prevented by using a JavaSymphony lock/unlock mechanism. If a thread t has a handle to an object and locks it, then no other thread can access this object until thread t unlocks it again. A lock operation is delayed until the currently unfinished object's methods terminate. These features are illustrated in the code excerpt from Fig. 3.17.

```
JSObject obj;
obj.lock(); //* lock object
... //* invoke methods of the JS object
obj.unlock(); //* unlock object
obj1.free() ; //* free object
```

Fig. 3.17. Lock/Unlock mechanism for JS objects. Code excerpt

Finally, an object, if no longer needed, should be released by invoking the *free* method, which reduces the overall bookkeeping effort and enables the garbage collector to de-allocate the memory used by the object. All the distributed objects used by a JS application are automatically released when the application un-registers from the JRS.

3.3.5 Persistent Objects

JavaSymphony provides facilities to make objects persistent by saving/loading them to/from external storage. An object can be stored in a local file, if none of its methods is currently executing, by using JSObject method *store*. Optionally, the file name is given; otherwise, a unique string is returned for the object that has just been stored (Fig. 3.18).

Objects are restored from local files, by invoking the static method *load* with the *str* string parameter that uniquely identifies a previously stored object. Optionally, parameters for the destination VA and single-/multithreaded attribute are used.

```
JSObject obj; String fName;

     // *** save object on external storage
fName = obj.store([givenName]);
...
     // *** load object from external storage
     // *** optional params: the target VA, single/multi-threaded attribute
JSObject obj = JSObject.load(str [,va] [,singleTh]);
```

Fig. 3.18. Using persistent JS objects. Code excerpt

3.4 Distributed Synchronization Mechanisms in JavaSymphony

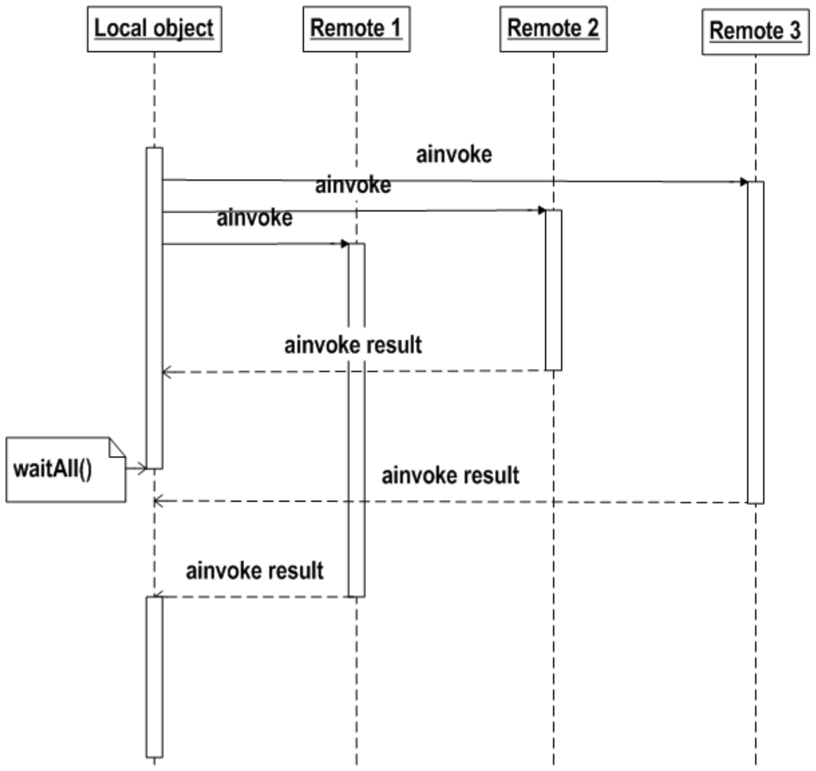


Fig. 3.19. Synchronization of 3 ainvoke calls

Synchronizing distributed threads of execution that are running in parallel is an important feature, which distributed systems should provide. We have already mentioned two synchronization mechanisms in JavaSymphony:

- Single-threaded objects, which restrict concurrent access and enforce sequential access to their methods.
- (Un)Lock/Unlock mechanism, which enables exclusive access to an object.

JavaSymphony adds further features that allow synchronization of multiple distributed objects. These features are described in the next sections.

3.4.1 Synchronization of Asynchronous Method Invocations

In concurrent systems, programmers commonly synchronize a set of threads or processes. In the presence of asynchronous method invocations, we found numerous cases, in which a set of threads that execute methods simultaneously – possibly on distinct computing nodes – should be synchronized in a join operation.

For this purpose, JavaSymphony enables the programmer to group a set of result handles, each one associated with a unique asynchronous remote method invocation, by using the *ResultHandleGroup* class. This class provides several methods to block or examine (without blocking) whether one, a specific number, or all the threads have finished processing the associated methods. Commonly, methods of distinct object, which may reside on distinct computing nodes, are executed in parallel in order to improve the load balancing and better utilize the existing computing resources. By using a synchronization mechanism, we can easily determine which object is idle because the execution of its method has finished. On the other hand, the returned results can be processed one by one at the time they are available, and the idle objects (i.e. for which the method invocation has finished) can be reused with a new asynchronous method invocation. Figure 3.19 and the code excerpt in Fig. 3.20 demonstrate this mechanism.

3.4.2 Barrier Synchronization

JavaSymphony provides a barrier mechanism that can be used to synchronize remote distributed threads of execution. A set of distributed threads, executing methods of JS objects may be suspended until all of them reach a certain barrier point (see Fig. 3.21). Once all threads have reached the barrier point, they resume the execution simultaneously.

The mechanism is straightforward, but improper usage can lead to performance degradation or even deadlocks. For each JS application, a set of barriers can be defined by using the *newBarrier* static method of *JSRegistry* class. This method has as parameters an identifier that uniquely identifies the barrier, and the number of the threads n that have to wait at barrier point.

```
JSObject obj[n];
ResultHandleSet rhs;
ResultHandle rh;
Object[] params;
...
for(i=0; i < n; i++) {
        //* add a ResultHandle and index i (optional) to the ResultHandleSet rhs
    rhs.add(obj[i].ainvoke("run", params), i);
}
...
if( rhs.isReady(5) ) {...} //* non-blocking test if at least 5 methods are finished
if( rhs.isAllReady() ) {...} //* non-blocking test if all methods are finished
if( rhs.waitReady(5) ) {...} //* block until at least 5 methods are finished
if( rhs.waitAll () ) {...} //* block until all methods are finished
    //* get results one by one without specific order;
    //** block until the first method has returned results
rh = rhs.getFirstReady();
while(rh ! = null)
{
    ResultClass result = (ResultClass)rh.getResult(); //* get the results
    ... //* process results;
    index = rhs.getIndex(rh); //* get index of idle object
        //* invoke next method on idle object for load balancing;
        //** add ResultHandle to the ResultHandleSet again
    rhs.add( obj[index].ainvoke("run", params), index);
        //* get ResultHandle of a method that finishes next;
        //** block until the method has returned results
    rh = rhs.getNextReady();
}
```

Fig. 3.20. Synchronization of ainvoke calls. Code excerpt

The barrier is visible to all the application objects. Upon reaching a barrier point, the execution threads are suspended until n threads have reached it. Thereafter, all threads may resume execution (Fig. 3.21). The code excerpt in Fig. 3.22 demonstrates the usage of the JavaSymphony barrier mechanism. A practical use of this mechanism is presented in Section 7.2.2.

3.5 JavaSymphony Distributed Events

Programs that incorporate objects reacting to a change of state somewhere outside the objects, possible on a different computing site, are common in both single address space and distributed systems. Commonly, user or system actions are modelled as events to which other objects in the program react. The events also represent a mechanism for asynchronous communication. Java

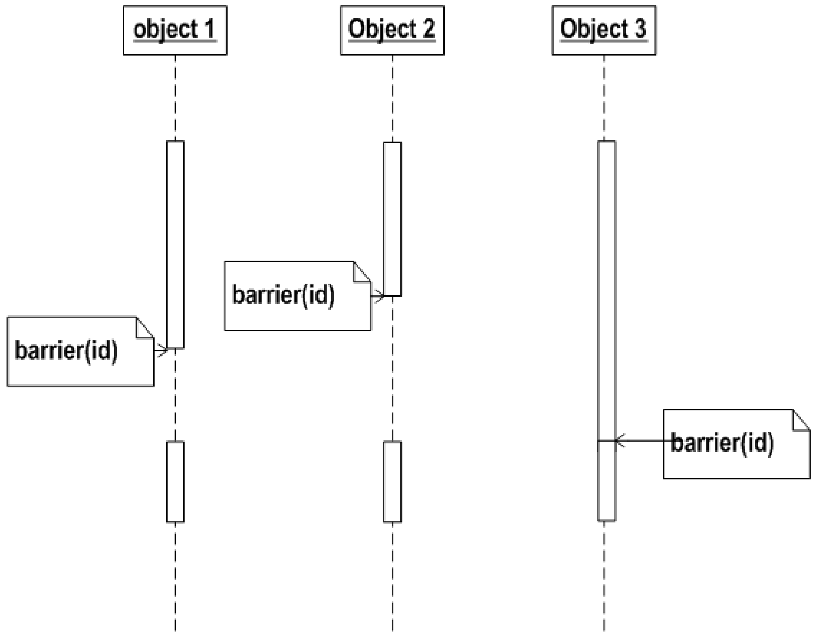


Fig. 3.21. Barrier synchronization for 3 JS objects

```
    //* ***** a barrierId defines a unique synchronization point *****
int barrierId = 17;
    //* define a barrier for two (remote) threads.
JSRegistry.newBarrier(2, barrierId);
obj1.oinvoke("runThread1", params);
obj2.oinvoke("runThread2", params);
...
    //* ***** inside runThread1 *****
int barrierId = 17;
    //* suspend execution until runThread2 reaches the synchronization point
JSRegistry.barrier(barrierId);
...
    //* ***** inside runThread2 *****
int barrierId = 17;
    //* suspend execution until runThread1 reaches the synchronization point
JSRegistry.barrier(barrierId);
...
```

Fig. 3.22. JavaSymphony barrier synchronization. Code excerpt

has a number of event models, differing in various subtle ways. All of these involve an object generating an event in response to some change of state either in the object itself or in the external environment. At some earlier stage, an event consumer would have registered interest in this event. A suitable consumer method will be called, in case of event occurrence.

JavaSymphony follows a general event model where objects can subscribe as consumers for various types of events. At a later stage, events of a specific type may be produced, in which case the corresponding registered consumers will be notified. An event consumer handles the event by providing an appropriate method. An advantage of the event mechanism is that JavaSymphony does not restrict the types of objects that receive or produce events. Any Java object distributed by using JavaSymphony can consume or produce events without implementing dedicated interfaces or extending dedicated JavaSymphony classes.

JavaSymphony supports three types of events:

- **User Defined Events** are explicitly generated by the user. They are used to support asynchronous communication and interaction among arbitrary Java objects (not restricted to JS objects). The programmer's tasks are to simply insert the specific code that generates an event and to provide a consumer's method that is invoked at the time the event notification arrives.
- **Middleware Events** are produced and controlled by the JRS, in the event of, for instance, VA unavailable, registration/un-registration of a new application, lock/unlock of JS objects or VAs, etc. The user must provide only the method that is invoked when the event notification arrives, whereas the JRS produces these events automatically.
- **System Events** are generated due to changes in the dynamic system parameters such as idle time, available memory, swap space allocated, etc. The programmer can access all of these parameters through the JavaSymphony API for static/dynamic system parameters (see Section 3.2.1). For the consumer object, a set of constraints, a specific constant value that controls the generation of an event, and a method, which is invoked if the event occurs, are specified as event consumer's constructor parameters. The event generation is triggered in one of the following 3 cases, depending on the constant value: (1) if the constraints hold (and previously did not), (2) if they do not hold, respectively (3) if they change their state. The JRS monitors the resources and examines whether the event must be generated or not.

The Event Consumer

An object that intends to consume a user-defined or JS-middleware event creates a *JSEventConsumer* that describes the event type (middleware or user-defined) and properties of the event in which it is interested. The *JSSystemEventConsumer* derived class is used in case of system events. When building an instance of the *JSEventConsumer*, the programmer provides the following information:

- Reference to the object that consumes the event;
- Unique event type identifier;
- The consumer's method, which will be invoked if the event occurs.

```
...
  //* define types for user defined and middleware events
int userEvType = JSConstants.C_USER_TYPE + 1;
int middleEvType = JSConstants.C_APP_REGISTERED;
JSObject listObj[]=.....; //* list of remotes objects
VA listVAs[]=.....; //* list of VAs
JSConstraints constr1;

  //* subscribe for the user-defined events;
  //** no restriction on event producers; handleMethod manages the events.
JSEventConsumer cEv1 = new JSEventConsumer(this, userEvType,
     JSConstants.C_ANY_LOCATION, "handleMethod");
  //* events can be produced only onto VAs in listVAs
JSEventConsumer cEv2 = new JSEventConsumer(this, userEvType,
     JSConstants.C_LIST_VA_EVENT, listVAs, "handleMethod");
  //* event can be produced only by JS Objects in listObj
JSEventConsumer cEv3 = new JSEventConsumer(this, userEvType,
     JSConstants.C_LIST_JSOBJECT_EVENT, listObj, "handleMethod");

  //* subscribe for a middleware event which can be produced anywhere;
  //** the event is generated when a new application registers with JS
JSEventConsumer cEv4 = new JSEventConsumer(this, middleEvType,
     JSConstants.C_ANY_LOCATION, "handleMethod");

  //* subscribe for a system event which is produced only by the VA va,
  //* if the validity for a set of constraints constr changes
JSSystemEventConsumer cEvSystem =
     new JSSystemEventConsumer(this, JSConstants.C_VA_EVENT,
     va, "handleMethod", constr, JSConstants.JS_CONSTRAINTS_CHANGE);
...
  //* subscribe for event cEv1
cEv1.subscribe();
...
  //* unsubscribe event cEv1
cEv1.unsubscribe();
...
```

Fig. 3.23. JavaSymphony event consumer. Code excerpt

The constants corresponding to the event types are defined as part of the *JSConstants* class. For example, C_USER_TYPE indicates the definition of user-defined events; C_APP_REGISTERED denotes an event generated when a new application registers itself with the JRS; C_SYSTEM_EVENT corresponds to system events.

Furthermore, the events can be filtered based on the event producer, by using specific parameters that are passed to the *JSEventConsumer* constructor. For instance, event producers can be limited to a list of producers (JS Objects) or to a producer that resides onto a specific location (i.e. VA). Similarly, a specific set of constants is used in these cases. For example, C_ANY_LOCATION indicates that the consumer accepts events from any producer; C_LIST_VA_EVENT restricts the location of the event producer to a specific list of VAs; C_LIST_JSOBJECT_EVENT restricts the event producer to a list of JS Objects.

For the consumers of system events, a *JSConstraints* object encapsulates the constraints that will be checked in order to produce a system event. For each set of constraints, JRS generates a distinct system event, which causes all the consumers to be notified accordingly. The generation of the event is controlled by an additional parameter, which can take one of the following three values:

- JS_CONSTRAINTS_HOLD - the event is generated if the constraints become valid;
- JS_CONSTRAINTS_NOT_HOLD - the event is generated if the constraints become valid;
- JS_CONSTRAINTS_CHANGE - the event is generated if the constraints status changes;

The utilization of event consumers is illustrated in Fig. 3.23. The consumer subscribes to an event by using the *subscribe* method of the *JSConsumerEvent* class. If event notification should no longer be received, then the consumer uses the *unsubscribe* method.

The Event Producer

Only user-defined events can be explicitly produced, by using the *JSEvent-Producer* object. The first parameter of the *JSEventProducer* constructor indicates the object that generates an event. The second parameter refers to the unique type of the generated event, which must match with the second parameter of the *JSEventConsumer*. Moreover, the constants used by event consumers to filter events based on event producer, are also used here to restrict the list of potential consumers. The user-defined event must be explicitly produced by invoking the *produceEvent* method of the *JSEventProducer*. The parameters passed to this method are forwarded to the consumers' methods by the JRS.

```
...
int userEvType = JSConstants.C_USER_TYPE + 1;
Object listObj[]=.....; //* list of remotes objects
Object listVAs[]=.....; //* list of VAS

  //* produces a user-defined event of type userEvType;
  //*** no restriction on event consumers
JSEventProducer pEv1 =
     new JSEventProducer (this, userEvType, JSConstants.C_ANY_LOCATION);
  //* notify only those consumers registered on VAs in listVAs
JSEventProducer pEv2 = new JSEventProducer (this, userEvType,
     JSConstants.C_LIST_VA_EVENT, listVAs);
  //* notify only those consumers in listObj
JSEventProducer pEv3 = new JSEventProducer (this, userEvType,
     JSConstants.C_LIST_JSOBJECT_EVENT, listObj);
...
  //* produce a user-defined event; parameters will be transmitted
  //* to the handleMethod of matching consumer
Object params[]=.....;
pEv1.produceEvent(params)
```

Fig. 3.24. JavaSymphony event producer. Code excerpt

Figure 3.24 shows code for event producer. We have also built an experimental application that uses JavaSymphony events, which is presented in Section 7.2.2.

3.6 Summary

JavaSymphony offers a new object oriented programming paradigm in Java, to control distributed and parallel system at a higher level. On the one hand, JavaSymphony supports automatic mapping, load balancing, and migration of objects without involving the programmer. However, fully automatic systems commonly cause poor performance results due to lack of information about the application and insufficient static and dynamic analysis. JavaSymphony, therefore, provides a semi-automatic mode, which leaves the error-prone, tedious, and time consuming low-level details (e.g. creating and handling of remote proxies for Java/RMI) to the underlying system, whereas the programmer controls the most important strategic decisions at a very high level, such as:

- The setup of the virtual distributed architecture by determining which computing resources can be used and how these resources should be organized for executing a distributed/parallel program.

- The mapping of data in relation to other data. For example, a set of objects may be placed physically close to each other or even on the same node of a VA if they intensively interact.
- The mapping of data (objects) onto specific nodes based on system constraints (e.g. available memory is larger than a minimal value, or CPU load is less than a maximal value).
- Dynamic conversion of conventional Java objects to JS objects, which enables access to conventional Java objects through JS remote method invocations. Moreover, JS objects can be dynamically modified to become single- or multi-threaded JS objects.
- Selective placement of code (Java byte-code) on specific computing nodes, which reduces the overall disk and memory requirement of an application.

In this chapter, we have introduced the JavaSymphony programming paradigm. The JavaSymphony programming API has two key components:

- **Dynamic virtual distributed architectures** are used to manage heterogeneous distributed computing resources and to control mapping, load balancing, migration of objects and code placement (Section 3.2);
- **JavaSymphony objects** are used to distribute code and data among remote computing resources, and to remotely execute code as part of a distributed application (Section 3.3).

In addition, JavaSymphony offers a large set of programming elements, which includes: various types of remote method invocation (e.g. synchronous, asynchronous and one-sided); lock/unlock mechanism for VAs and JS objects; high level API to access a large variety of static or dynamic system parameters, selective remote class-loading; automatic and user-controlled mapping of objects; conversion from Java conventional objects to JS objects for remote access; single-threaded versus multi-threaded JS objects; object migration, automatic or user-controlled, etc. Moreover, JavaSymphony introduces high-level distributed synchronization mechanisms (Section 3.4) and distributed event mechanism (Section 3.5).

4
JavaSymphony Runtime System

The JavaSymphony programming paradigm offers high-level constructs to simplify the programming for distributed systems. The JavaSymphony class library is used to distribute data and to run code on a collection of computers. In order to use a set of distributed computing machines for a distributed application, the JavaSymphony middleware is required to run on each of them.

In this chapter, we describe the functionality of the JavaSymphony middleware and present several implementation issues. The chapter is organized as follows: The first section gives an overview of the JavaSymphony Runtime System (JRS). Thereafter, we analyze the main components of the middleware:

- **The JavaSymphony Administration Shell** (Section 4.2) is a centralized graphical tool used to manage the set of available computing resources.
- **The Network Agent System** (Section 4.3) monitors and manages each computing resource.
- **The Object Agent System** (Section 4.4) manages the interaction between the JavaSymphony middleware and the JavaSymphony distributed applications.
- **The Event Agent system** (Section 4.5) manages the JavaSymphony distributed event mechanism.

Finally, Section 4.6 presents a formal model of the JavaSymphony Runtime System based on the Pi-calculus.

4.1 Runtime System Overview

The JavaSymphony Runtime System (JRS) is implemented as an agent based system (see Fig. 4.1) that consists of several components: the JavaSymphony Administration Shell (JS-Shell), the Network Agent System (NAS), the Object Agent System (OAS) and the Event Agent System (EvAS). The Network

Agent System is made up of network agents (NAs), which run on every machine (PC/workstation, SMP node, supercomputer) that can be used to start or to run a JS application. The Object Agent System comprises two types of object agents: public object agents (PubOA), one for each computing node, and application object agents (AppOA), one for each JS application. The Event Agent System includes event agents (EvA), one for each machine to be used by the JS applications.

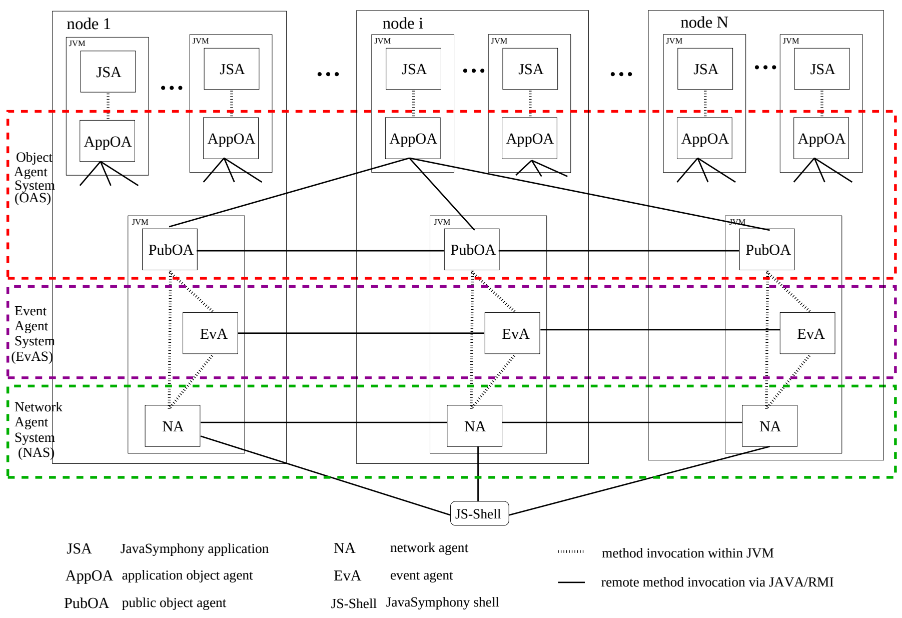

node 1
node i
node N
JVM
JSA
AppOA
PubOA
EvA
NA
JS-Shell
Object Agent System (OAS)
Event Agent System (EvAS)
Network Agent System (NAS)
JSA JavaSymphony application
AppOA application object agent
PubOA public object agent
NA network agent
EvA event agent
JS-Shell JavaSymphony shell
method invocation within JVM
remote method invocation via JAVA/RMI

Fig. 4.1. JavaSymphony Runtime System (JRS) Architecture

Each machine to be used by JavaSymphony has a NA, a PubOA and an EvA running in the same JVM, while each JS application runs in its own JVM instance together with its associated AppOA.

4.2 The JavaSymphony Administration Shell (JS-Shell)

All physical computing resources (nodes) such as workstations, PCs, clusters or supercomputers must have a JVM installed and must be configured under the JS-Shell (see Fig. 4.2) before they can be used by any JS application. A NA can be started on each configured node in two ways:

- Automatically by the JS-Shell. The JS-Shell spawns NA processes onto a predefined set of the machines within the same administrative domain, provided that the domain security policy allows the JS-Shell process to remotely start these processes within the domain. In a Grid environment, the security issues can be solved by the Grid security infrastructure: JS-Shell may use the Grid resource manager (e.g. Globus toolkit [12]) to allocate resources and start NAs onto them.

- Manually, by the user, on each machine that is to be used in distributed calculations by JavaSymphony applications. JS-Shell process provides an access point (IP address and port), which can be accessed by newly started NAs, by using the Java RMI mechanism. The NAs contact the JS-Shell, which adds them to its list of managed resources.

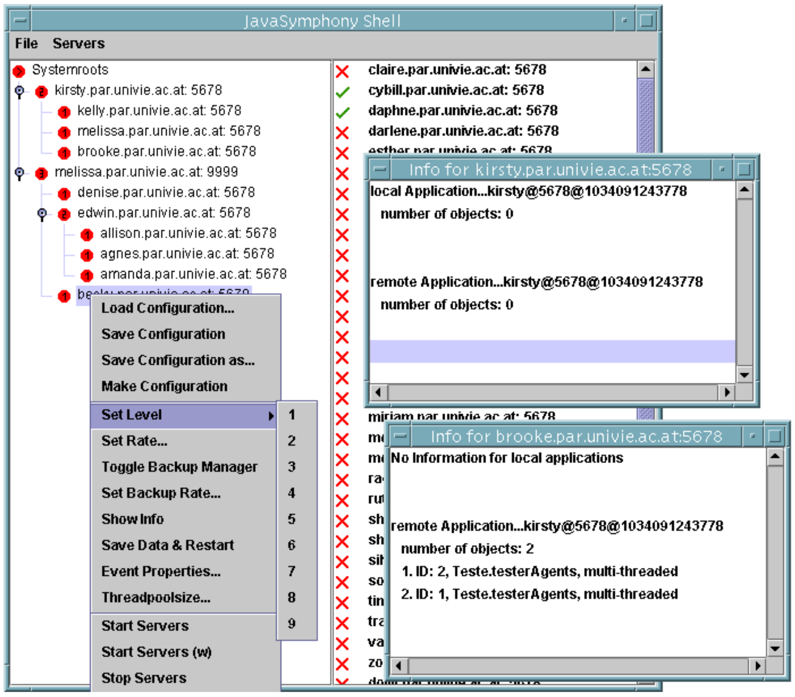
Computing resources can be dynamically added or eliminated. The JS-Shell controls the NAs via the Java/RMI mechanism. Using resources in multiple administration domains can complicate the communication between distinct NAs, respectively between the NAs and the JS-Shell, especially if security policies are enforced (e.g. firewalls restrict access to or even hide the address of computers in the domain). These security issues can be solved by using additional JS proxies processes, which reroute the low-level socket communication between JavaSymphony agents through secure channels that can bypass the firewalls. In this case, additional settings (e.g. proxy address, proxy port, list of acceptable remote partners) are stored in local files at each machine that runs a NA.

The computing resources are monitored using the JS-Shell GUI. The runtime behaviour of the JS applications can be observed in the node information window. The remote objects of an application distributed on several nodes are listed in the same window and can be migrated to a remote node using this interface. For example, the right side of Figure 4.2 shows the monitoring windows for two nodes (workstations) named *kirsty* and *brooke*. On both machines a NA is running. A JS application is started on *kirsty* and appears in its monitoring window as a local application. This application creates 2 remote objects onto *brooke*, which are shown in its monitoring window.

The JS-Shell interface can be also used to configure various parameters of the JRS (e.g. number of JobHandlers, enable/disable specific system events, backup in case of failure of resources, etc.). Predefined resource configurations may be stored into files or loaded from files. These configurations may be used by the JS-Shell to automatically start up or stop large sets of NAs onto the corresponding machines.

4.3 The Network Agent System (NAS)

A network agent (NA) is started by the JS-Shell or independently by the user, onto each computing node to be used by the JS applications. The NA determines the local machine's setting (e.g. local temporary directory, proxy settings for communication among multiple administrative domains, local port used to access the NA, etc.) from a specific local file, and starts one associated PubOA and one associated EvA. The NA's main purpose is to monitor



```
Systemroots
  kirsty.par.univie.ac.at: 5678
    kelly.par.univie.ac.at: 5678
    melissa.par.univie.ac.at: 5678
    brooke.par.univie.ac.at: 5678
  melissa.par.univie.ac.at: 9999
    denise.par.univie.ac.at: 5678
    edwin.par.univie.ac.at: 5678
      allison.par.univie.ac.at: 5678
      agnes.par.univie.ac.at: 5678
      amanda.par.univie.ac.at: 5678
    be
```

```
claire.par.univie.ac.at: 5678
cybill.par.univie.ac.at: 5678
daphne.par.univie.ac.at: 5678
darlene.par.univie.ac.at: 5678
esther.par.univie.ac.at: 5678
miriam.par.univie.ac.at: 5678
```

```
Info for kirsty.par.univie.ac.at:5678

local Application...kirsty@5678@1034091243778
  number of objects: 0

remote Application...kirsty@5678@1034091243778
  number of objects: 0
```

```
Info for brooke.par.univie.ac.at:5678

No Information for local applications

remote Application...kirsty@5678@1034091243778
  number of objects: 2
  1. ID: 2, Teste.testerAgents, multi-threaded
  2. ID: 1, Teste.testerAgents, multi-threaded
```

Fig. 4.2. JS-Shell, an interface to control the physical resources for JS applications

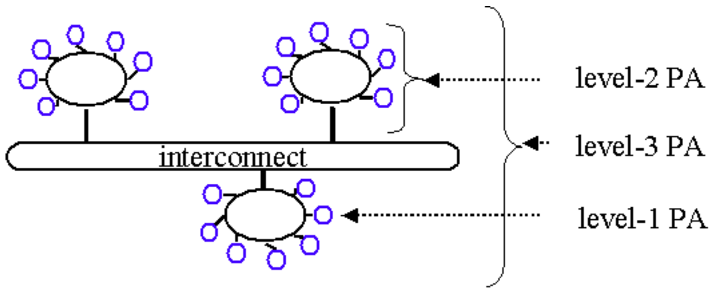


Fig. 4.3. Example of a level-3 PA (physical architecture)

the resources of the local machine (e.g. dynamic parameters as system load, free memory or static parameters as operating system, machine name, Java version). At the same time, the NA monitors the constraints used to create the VAs, or used to create or migrate JS objects, and chooses the machines that build up the requested VAs.

The set of all NAs defines the Network Agent System (NAS). At the level of the NAS, the computing nodes are organized in a level-i ($i \geq 1$) tree structure,

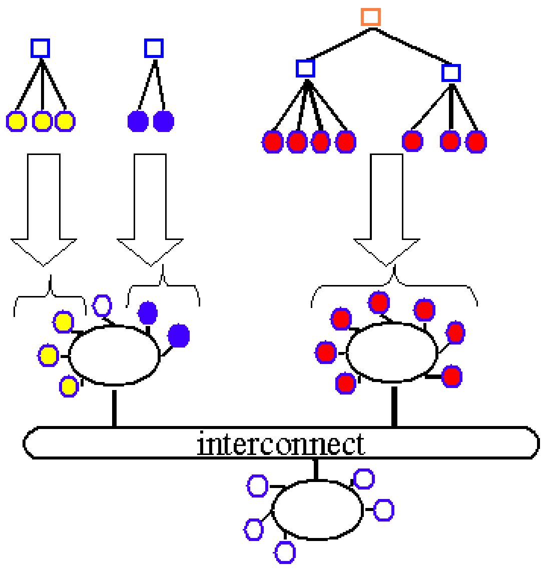


Fig. 4.4. Mapping level-2,3 VAs to level-2 PAs

which we call physical architecture (PA) (Fig. 4.3). A level-2 PA represents a cluster and consists of a set of nodes (level-1 PA). A level-3 PA represents a cluster of clusters, etc. A PA manager, which is a node as well, controls the nodes in a PA. In addition to the management tasks, a PA manager can also be used as a computing node for JS applications. The left side of Figure 4.2 shows a complex PA, which is built by using the JS-Shell.

The VAs used by the JS applications map their nodes (level-1 VAs) onto the nodes in a PA. Higher-level VAs do not correspond to the nodes in a PA. Assuming that a level-n VA with k nodes (the VA tree structure has k leaves) is requested by a JS application, the nodes are chosen such as the best locality is provided, according to the following algorithm:

1. If a cluster (level-2 PA) that includes at least k nodes (which fulfil the constraints associated to the given VA) is found, then all VA's nodes are chosen from this cluster (Fig. 4.4).
2. If no corresponding level-i PA ($i >= 2$, initially i is 2) is found, then the algorithm recursively searches for level-$(i - 1)$ PAs for each level-$(n - 1)$ sub-VAs. The result is the aggregation of all these sub-VAs (Fig. 4.5).
3. If no PA is found, then the algorithm increments i and repeats the previous step.

The PA manager periodically monitors each node in its PA subtree and collects statistical values for the system parameters. The dynamic values are updated periodically, according to the settings controlled by the JS-Shell. A level-I PA manager collects the observed system parameters of its level-$(i-1)$

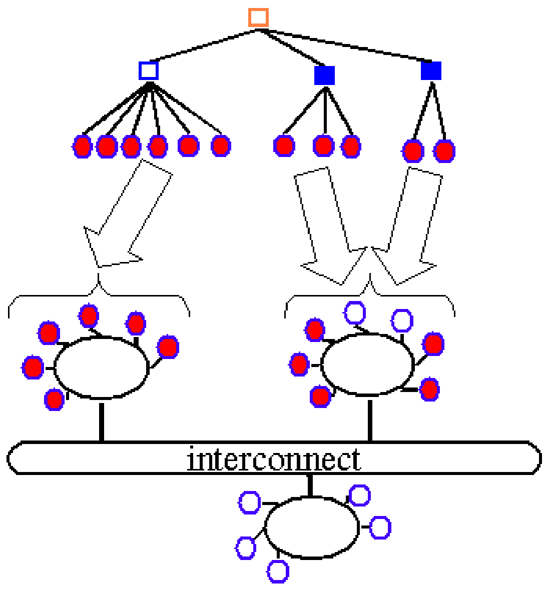


Fig. 4.5. Mapping a level-3 VA to a level-3 PA

PAs, locally computes and stores simple statistics, and forwards them to its associated level-$(i + 1)$ PA. Thus every PA has statistical information about the system behaviour for all its architecture components at the next lower level.

Monitoring is also used to detect system failures. If a certain node does not respond within a specific length of time, it will be released by the NAS according to a simple fault tolerance mechanism.

4.4 The Object Agent System

The Object Agent System (OAS - see Fig. 4.1) directly interacts with JS applications by managing the remote objects (e.g. creation, mapping, migration, load balancing, and release of JS objects). Furthermore, the OAS is responsible for the management of the remote method invocations (e.g. parameter transfer, remote execution, and returning results to the calling site).

Each NA creates a public object agent (PubOA) within the same JVM. For each JS application, a unique application object agent (AppOA) is generated once the JS application registers with the JRS. A JS application uses the same JVM as its associated AppOA. PubOAs and NAs on the one hand, and AppOAs and JS applications on the other hand interact by local (direct) method invocations. The Java RMI mechanism is used for communication between the AppOA and PubOAs. The AppOA manages a list that contains all the objects generated by its corresponding JS application.

If a JS application requests a VA, the associated AppOA forwards this request to the PubOA. The *VAManager*, as part of the PubOA, interacts with the local NA and manages (creates, modifies, and releases) all VAs that have been asked by any JS application on the node where the PubOA resides.

Job Java Class	Job Description
ChangeObjectLockJob	Job to (un)lock a (remote) JS object
ChangeObjectTypeJob	Job to change the type of an JS object (e.g. single- or multi-threaded).
CreateObjectJob	Job to create a new JS object/ load it from an external storage
DownloadCodebaseJob	Job to download the codebase for an object, from a remote node; used in migration
DownloadObjectJob	Job to download a remote object (for storing into an external storage)
FreeCodebaseJob	Job to free codebase from the node
LoadCodebaseJob	Job to install the necessary codebase on remote nodes
MethodInvocationJob	Job to invoke methods of (remote) JS objects
TransferObjectJob	Job to migrate an object to a remote location

Table 4.1. Job types in JavaSymphony

A *job processing mechanism* (Fig. 4.6) is used to implement any type of remote and local interaction between OAs. A *job* may create a JS object, invoke a method, download a codebase, migrate objects, lock/unlock objects or VAs, etc. Table 4.1 lists the JavaSymphony job types with the description of their tasks.

Each PubOA and AppOA creates a set of threads called *JobHandlers*, which process all the jobs submitted remotely by using Java/RMI, or locally through direct method invocation. The number of *JobHandler* threads is controlled via the JS-Shell. For the single-threaded JS objects, a dedicated *JobHandler* thread is created, whereas for the multi-threaded objects (default case), several threads can be employed to simultaneously process jobs.

We demonstrate the job mechanism functionality for the *MethodInvocationJob*: For every method invocation in a JS application which implicates local or remote OA processing, a *MethodInvocator*, part of the OA, creates a specific job (Fig. 4.6). If the job implies receiving results (e.g. synchronous and asynchronous method invocations), a *ResultHandle* is created and attached to

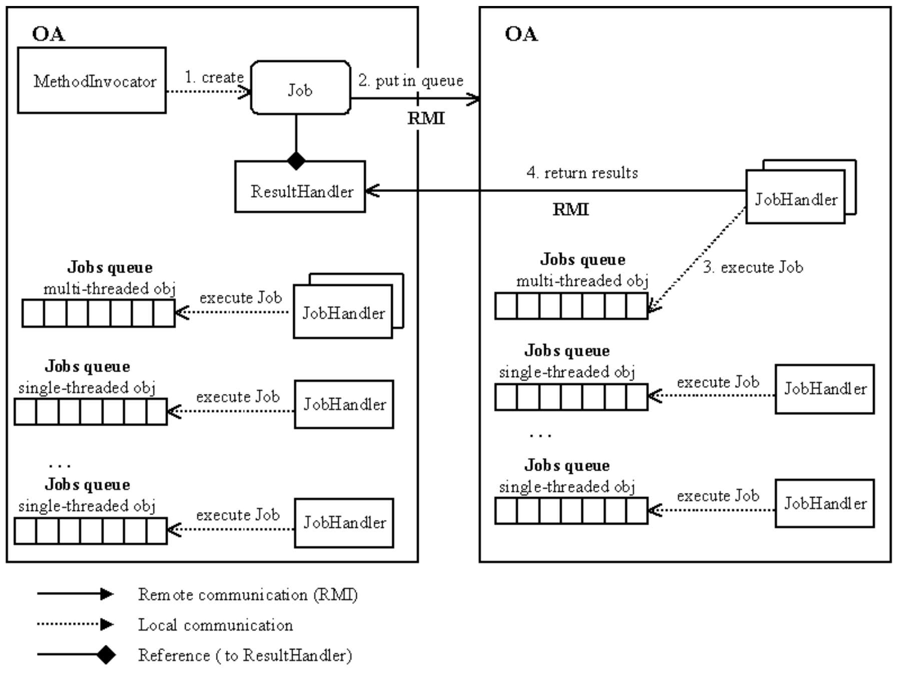


Fig. 4.6. Job processing mechanism

the job. The OAs and the *ResultHandle* play the role of the remote objects in the Java RMI mechanism [20], whereas the jobs, respectively the results, are serializable objects that are passed as parameters to the remote OA's methods, and respectively to the remote *ResultHandle*'s method. Depending on whether they cause method invocations for either multi-threaded, or for single-threaded objects, the jobs are placed into corresponding queues at the destination OA. After the object method is executed, the available results are sent back to the associated *ResultHandler* via Java RMI.

In the following, we discuss how the JRS manages and supports some of the most important JavaSymphony programming features.

VA Management

If a JS application requests a VA, the associated AppOA forwards this request to the *VAManager* of the local PubOA. The *VAManager* manages all VAs generated by any JS application that runs onto the local node. The following information is stored for each VA: VA identification and description, identification of the associated JS application and AppOA, constraints that must hold for this VA. The *VAManager* periodically examines whether the constraints of all stored VAs still hold, by accessing system parameters via its local NA. In the current implementation, the JS programmer must decide which action must be taken if the constraints are no longer fulfilled.

Object Creation

Remote parallel processing in JavaSyphony is based on remote objects. For each created JS object, the JS application obtains a handle, which is managed by the associated AppOA. Each JS object creation is done by a specific job: *CreateObjectJob*. The job is sent to the associated AppOA, if the object resides locally, onto the node where JS application was started, or to a remote PubOA, if the object is placed at another location. A *JobHandler* executes the job. A newly created object is added in the AppOA's local list of JS objects. The JS programmer can fully control the mapping by indicating a specific VA, or by specifying which system constraints must be fulfilled by the VA where the object will be placed.

Method Invocation

Each AppOA/PubOA that possesses an object handle has full access to the object's methods via its *MethodInvocator*. Both method's name and parameters are encapsulated in a serializable *MethodInvocationJob* object, which is transmitted to the PubOA or AppOA where the object resides. Object handles are associated with information about the location of the object and the AppOA/PubOA which the object originates from. Object handles are first-order objects, thus having the possibility to be passed to methods of other objects that may reside onto arbitrary nodes. Methods are always executed by the local AppOA or by the remote PubOA, corresponding to the location where the object has been generated. The results are sent back to a *ResultHandler*. Note that *ResultHandlers* are employed for both local and remote method invocations.

Object Migration

The JS application programmer can explicitly migrate objects to other VAs. An object to be migrated is serialized and transferred by the OAS to a different location/VA. Moreover, the AppOA's list of objects (associated with the JS application) and the list at the remote PubOA (where the object will reside after migration) are updated to reflect the change of location. Explicit methods are provided by the JS programming API to examine whether initial constraints indicated during creation of a JS object still hold or whenever any system parameter reaches a certain threshold. The remote references to the migrated object are not instantly updated with the new location. If a method invocation tries to find an object at the old location, an exception is thrown. Thereafter, the object's new location can be found at the AppOA associated with the JS application (home location). Note that each JS application is associated with a unique AppOA, but possibly with several/many PubOAs (see Fig. 4.1).

Lock/Unlock

The JS programming API allows a programmer to explicitly lock and unlock JS objects. If a thread acquires a lock for a JS object, all future jobs related to the object are placed into a separate queue and delayed for execution, until the object is unlocked. Only the thread that has initialized the lock is allowed to unlock it. Until this happens, only jobs "signed" by this thread may access the regular job queue, and can be processed by the *JobHandlers*. Once the unlock operation occurs, the blocked jobs are transferred back into the regular job queue.

Codebase Transfer

Usually the codebase transfer is done during the application initialization phase, when all necessary code is placed onto the computing nodes in the VAs. Additional transfers may be needed when migrating the objects or creating new VAs. The codebase content (e.g. classes and archives) is serialized and transferred to the destination. The target PubOA deserializes the content of the associated jobs and places it into the local classpath, into a temporary directory.

4.5 The Event Agent System (EvAS)

The Event Agent System (EvAS) consists of the event agents (EvA) that run on all the machines used by JS applications. Onto each of these machines, the NA starts an associated EvA. An EvA directly interacts with the local PubOA to register user-defined events, middleware events, and system events consumers or to produce user-defined events (see Section 3.5). The EvA also interacts directly with the local NA to produce system events. The middleware events are configured/selected by using the JS-Shell and are automatically produced by JS classes.

The interaction between consumers and producers is realized through the EvAS (see Figure 4.7). To register for an event, a consumer (any object in a JS application) contacts the local EvA and provides information about the event it wants to receive (e.g. event type, filter for producers, constrains for system events, method to handle the event, etc. - see Section sec:event) . The EvA analyses the filter associated with the event consumer, determines the list of possible producers and distributes (remotely) the request to the corresponding EvAs. Each of these EvAs manages a list of registered consumers.

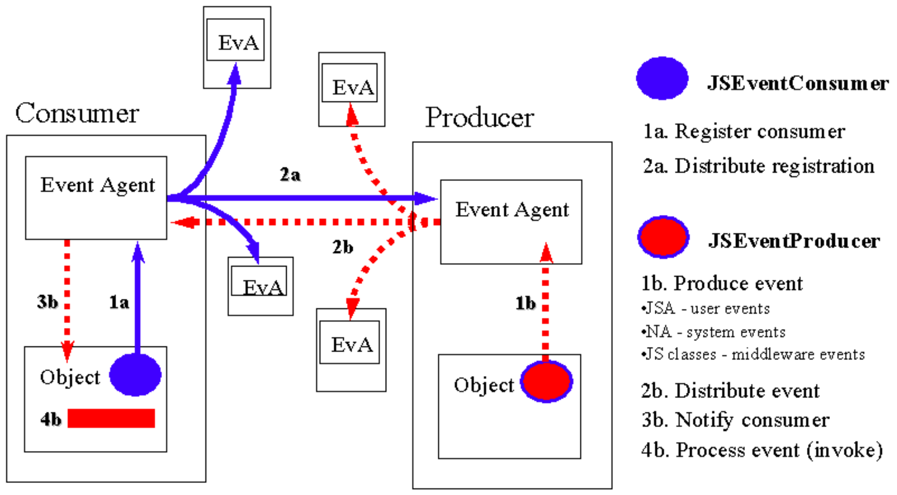


Fig. 4.7. Event Agent System. Consumer and producer interaction

In order to produce an event, an event producer (i.e. any object in the JS application for user events, JS classes for middleware events or a NA for system events) contacts the local EvA and provides information about the event (e.g. event type, a filter for the destinations, etc.). The EvA inspects the filter and the type associated with the event to find the list of possible consumers and (remotely) notifies the corresponding EvAs about the event's occurrence. The EvA at the consumer's location contacts the consumer by (locally) invoking its event handling method. There is no direct connection between the producer(s) and the consumer(s). Local interaction among EvAs and consumers, respectively producers, is done via local method calls. The RMI mechanism is used for remote interactions between distinct EvAs.

4.6 Modelling the JavaSymphony Runtime System with Pi-calculus Processes

Pi-calculus[1] [34] is the most recent addition to the impressive collection of process algebra variants. Pi-calculus can be seen as a minimal programming language built to capture all interesting behaviours of concurrent programs and gives us a mean of expressing the dynamic interactions among communicating processes, which makes it suitable to build a formal model for the functionality of the JRS. On the other hand, it is commonly thought that Pi-calculus is at a too low-level to be used as a serious tool.

[1] Preliminary notions of Pi-calculus are presented in Appendix 10.3

However, we believe that it would be useful to investigate at what extend Pi-calculus can be used to model a real world system - in our case the JRS. Therefore, we use a Pi-calculus variant (Appendix 10.3) to express the dynamic interactions between the components of the JRS: the JS-Shell, the NAS, the OAS, the EvAS and the JS applications.

The components of the model are presented in a top-down manner: First, we build a reduced formula for the whole system (i.e. the JRS), while ignoring the definitions of the subparts. Thereafter, we separately analyze the subparts (i.e. the JS-Shell, the NAS, the OAS, the EvAS and the JS applications).

4.6.1 The Runtime System

The JavaSymphony Runtime System can be represented as the parallel executions of 3 agent systems and a monitoring process:

$$JS ::= NAS|OAS|EvAS|JSShell$$

On the other hand, it can be represented as the sum of agents running on each computing resource:

$$JS ::= (\nu\widetilde{c})AG_1|AG_2|.....AG_n|JSShell$$

where $AG_x ::= (\nu\widetilde{c}_x)\{NA|PubOA|EvA\}_x$

We assume the existence of private channels (i.e. placed under restriction operator ν) for each pair of connected agents (according to Fig. 4.1). For example, there is a communication channel from $PubOA_i$ to $PubOA_j$, which we denote by $c_{PubOA_i-PubOA_j}$, and there is also a channel from $PubOA_j$ to $PubOA_i$, denoted by $c_{PubOA_j-PubOA_i}$. In the same manner, there are remote communication channels among NAs or among EvAs, between the AppOA and the PubOAs or between the JS-Shell and the NAs, for which we use similar notations. There are also local communication channels within one computing resource, such as $c_{NA_x-PubOA_x}$ or $c_{EvA_x-PubOA_x}$.

We use the notation $\widetilde{c}$ to represent the set of all these channels, whilst $\widetilde{c}_x$ is used to denote the set of channels that connect the agents inside a single computing resource x.

The mechanism of using input and output communication channels for each pair of connected agents is quite generic. On the other hand, we may want to model additional specific behaviour that may apply for distributed systems. For example, in our implementation, the agents within a single JVM are listening to the same port of the machine, which is allowed by the RMI lookup mechanism. We could represented this behaviour in Pi-calculus as:

$$(\nu c)c(x).NA(x)|PubOA(x)|EvA(x)$$

The protocol that chooses which agent process the input message (x) is complex and is hidden behind the definitions of the NA, PubOA and EvA. On the other hand, since the agents are implemented as RMI services, they are bound to specific names. Therefore, we can consider that the input communication channels are distinct as in the following formula:

$$(\nu c_{NA}, c_{PubOA}, c_{EvA})c_{NA}(x).NA(x)|c_{PubOA}(x).PubOA(x)|c_{EvA}(x).EvA(x)$$

Alternatively, we can use the matching construct of Pi-calculus and assume that the message x, which is coming over the channel c encapsulates information about the destination agent (e.g. the type of the agent). This can be expressed by the following formula:

$$\begin{cases} [x \mapsto dest = NA]NA(x) \\ [x \mapsto dest = PubOA]PubOA(x) \\ [x \mapsto dest = EvA]EvA(x) \end{cases}$$

4.6.2 RMI and Pi-calculus

The JavaSymphony Runtime System is built upon the Java RMI mechanism [20]. Pi-calculus operates with channels and communication over channels, which is at a lower level than the Java RMI mechanism. This is the reason for which we find useful to express the RMI mechanism in terms of Pi-calculus.

Typically, RMI applications have two separate parts: server(s) and clients. The server creates remote objects, makes them accessible and waits for some clients to invoke their methods. The clients get remote references to these objects to invoke methods on them. The Java/RMI mechanism allows the server and the client to communicate and to pass information.

We express this mechanism in terms of Pi-calculus.

- We use Pi-calculus process to model a remote object. The object is accessible by using the IP address of its location, the port where the *rmiregistry* is listening for calls and the name associated by the server to the object. Therefore, we consider that the object is accessible via the channel $c_{addr,port,name}$, which is uniquely determined by these parameters.
- The client obtains a remote reference which has a specific remote interface type. The interface type corresponds with the sort associated with the channel $c_{addr,port,name}$, which determines the specific data type allowed to be send along the channel.
- The client calls the method in a standard way. In Pi-calculus, this means that it sends the method (or method name) and the parameters along the channel: $\overline{c_{addr,port,name}}\langle method, params...\rangle$ ....
  Note that the sort associated with the channel restricts the methods and parameters according to the interface definition.
- The server executes the method on the remote object. This is modelled by a separate process: $Execute(obj \mapsto method, params)$, which hides the complexity of the method invocation.

- The client gets the result over a new channel, which it has provided at invocation. Again, the channel is associated with a sort that allows only the type of the expected result along the new channel.
- In summary, we can formalize the client and the server as:
  $Client ::= (\nu c_{result})\overline{c_{addr,port,name}}\,\langle method, params, c_{result}\rangle\,.c_{result}(x).P$
  $Server ::= !c_{addr,port,name}(method, params, c_{result}).$
  $Execute(obj \mapsto method, params).\overline{c_{result}}\,\langle result\rangle$
  P is the continuation of the client. The replication in the server's definition means the ability to serve multiple clients.

4.6.3 The Network Agent System

The Network Agent System comprises a set of NAs, running on each computing resource that is used by JavaSymphony:

$$NAS ::= (\nu\tilde{c})\{NA\}_1|\{NA\}_2 \cdots |\{NA\}_m$$

In JavaSymphony, a NA process acts as a RMI server. In addition, a NA has associated a backup-thread, which implements a simple fault-tolerance mechanism, and a property thread, which computes the static and dynamic system parameters. We can express this as:

$$NA ::= !(c_{NA}(m, p, c_{res}).Execute(m, p).\overline{c_{res}}\,\langle result\rangle)|PropTh|BackupTh$$

The property-thread collects statistical information from the successors in the resource tree (physical architecture - see Section 4.3), computes the system properties for its own system and forwards statistical information to the predecessor in the tree. This process is cyclic:

$$PropTh ::= c_{in}(i).ComputeProp(...).\overline{c_{out}}\,\langle o\rangle\,.PropTh$$

The backup-thread checks the parent of the current node. In case of failure, the process starts reorganizing the resource tree.

$$BackupTh ::= c_{check}(x).[x = fail]InitBackup(...).BackupTh$$

Note that the complexity of the actual NA is hidden behind the definitions of *Execute*, *ComputeProp* or *InitBackup*. It is not possible to model these processes with simple Pi-calculus formulas.

Also, we can see that *PropTh* and *BackupTh* are defined recursively, since they are cyclic processes. There is a waiting time before starting the next cycle of the process, which we do not represent in terms of Pi-calculus. An alternative is to insert a process $Wait(...)$, before starting the next cycle.

Another interesting aspect is the choice of the channels c_{check}, c_{in} (in fact an array of channels for the successors) and c_{out}. These channels are not placed under the restriction operator ν, because they are initialized by the

$Execute(m, p)$ by using a specific NA's initialization method m and specific parameters. This method is remotely invoked by the *JSShell* process, which contacts all the NAs (along the channels c_{NA}) and sets up the resources tree, before any application could run.

4.6.4 The Object Agent System

The Object Agent System (OAS) directly interacts with the distributed applications. The constructs of JavaSymphony programming paradigm are supported by operations at the OAS level. Consequently, the OAS is the most important and most complex agent system in JavaSymphony.

The OAS comprises two types of object agents:

- The public object agents (PubOAs). One PubOA is running onto each computing resource;
- The application object agents (AppOAs). One AppOA instance is created for each JavaSymphony application.

The following formula models the OAS:

$$(\nu\tilde{c})\,\{PubOA\}_1\,|...|\,\{PubOA\}_m\,|\,\{AppOA\}_{m_1}\,|\,\{AppOA\}_{m_2}\,|...|\,\{AppOA\}_{m_n}$$

An object agent (OA - either PubOA or AppOA) acts as an RMI server. In addition, several *JobHandler* threads are running under the control of each OA (see Fig. 4.6). Therefore, we model an OA as following:

$$\begin{aligned} OA ::= {} & (\nu in, out)(Q(in, out)| \\ & (c_{OA}(m, p, c_{res}).Execute(m, p).\overline{c_{res}}\,\langle result\rangle)\,| \\ & JH_1|JH_2|...JH_n) \end{aligned}$$

Again, the complexity of the OA is hidden behind $Execute(m, p)$ and JHs processes. JH_1, JH_2, ... JH_n are identical processes, corresponding to the *JobHandler* threads. The (finite) number of *JobHandler* threads is controlled via the JS-Shell.

Each of the *JobHandler* threads extracts a job from the OA's job queue and processes it. We assume that the queue is modelled by a process $Q(in, out) ::= in(x)....Q|\overline{out}\,\langle y\rangle\,....Q$, which reads from the *in* channel and writes to the *out* channel. The readings and writings are asynchronous, which is difficult to express in terms of Pi-calculus. Therefore, we do not further detail the queue process's formula.

The threads' behaviour is quite simple. One job is taken from the queue Q, its *run* method is executed and the result is sent back to the invocation source (see Fig. 4.6):

$$JH ::= out(job).Execute(job \mapsto run).\overline{job \mapsto output}\,\langle result\rangle\,.JH$$

The job types that are used in JavaSymphony are listed in Table 4.1. The jobs are placed in the queue by (remotely) invoking the OA's *putJob* method with the job as parameter. This can be written as:

$$\overline{c_{OA}}\left\langle pushJob, job, c_{result}\right\rangle .P|OA$$

where the first process sends $(pushJob, job, c_{result})$ over the c_{OA} channel and continues as P.

We assume that $Execute(pushJob, j) \rightarrow \overline{in}\left\langle j\right\rangle$, since this method is only placing the job in the queue. In parallel with OA, the process is transform by reduction as follows:

$$\begin{aligned}
&\overline{c_{OA}}\left\langle pushJob, job, c_{result}\right\rangle .P|(\nu in, out)(Q(in, out)| \\
&\quad !\,(c_{OA}(m, p, c_{res}).Execute(m, p).\overline{c_{res}}\left\langle result\right\rangle)\,|JH_1|JH_2|...JH_n) \\
&\rightarrow P|(\nu in, out)(Execute(pushJob, job)|Q(in, out)| \\
&\quad !\,(c_{OA}(m, p, c_{res}).Execute(m, p).\overline{c_{res}}\left\langle result\right\rangle)\,|JH_1|JH_2|...JH_n) \\
&\rightarrow P|\left((\nu in, out)\overline{in}\left\langle job\right\rangle|Q(in, out)|\,(...)\,|JH_1|JH_2|...JH_n\right) \\
&\rightarrow P|\left((\nu in, out)\overline{out}\left\langle job\right\rangle|\,(...)\,|JH_1|JH_2|...JH_n\right)
\end{aligned}$$

and

$\overline{out}\left\langle j\right\rangle|JH_i$ is $\overline{out}\left\langle j\right\rangle|out(job).Execute(job \mapsto run).\overline{job \mapsto output}\left\langle results\right\rangle$

which is reduced to

$$Execute(j \mapsto run).\overline{j \mapsto output}\left\langle results\right\rangle$$

4.6.5 The Event Agent System

The Event Agent System (EvAS) consists of a set of event agents (EvAs), running on each computing resource that is used by JavaSymphony:

$$EvAS ::= (\nu\tilde{c})\{EvA\}_1|\{EvA\}_2\cdots|\{EvA\}_m$$

An EvA acts as a RMI server in JavaSymphony. It performs several primary operations:

- Registration of a local consumer. A local object $Cons$ registers its interest in receiving remote events of a specific type TEv. The EvA processes the information about the consumer and the event of interest, determines a list of remote EvAs that may send this type of event, and remotely registers the consumer with each of these EvAs. We model this operation as:

  $c_{localC}(Cons, TEv).ProcessC(Cons, TEv).$
  $(\overline{c_{remoteC_1}}\left\langle Cons, TEv\right\rangle|\overline{c_{remoteC_2}}\left\langle Cons, TEv\right\rangle|...|\overline{c_{remoteC_n}}\left\langle Cons, TEv\right\rangle)$

  The formula shows that the process receives $Cons$ and TEv along a dedicated channel c_{localC}, process this information in $ProcessC$ sub-process, which determines the correspondent remote EvAs and the associated channels $c_{remoteC_i}$. $Cons$ and TEv are sent over these channels. $ProcessC$ also saves information about the consumer into local memory.

- Registration of a remote consumer. An EvA determines which remote EvAs may produce events of interest for a local consumer. Afterwards, it sends information about the producer and the event of interest to these remote EvAs. We write this as:

$$c_{remoteC}(Cons, TEv).RegisterC(Cons, TEv)$$

  The *RegisterC* sub-process places the information about *Cons* and *TEv* into a local list. The channel $c_{remoteC}$ is one of the $c_{remoteC_i}$ channels mentioned above.

- Local event notification. A local event producer generates an event and sends a notification to the local EvA. The EvA checks its list of registered consumers and determines which remote EvAs need to be notified. A notification is sent to these EvAs. The formula is similar to the formula for local consumer registration:

$$c_{localP}(Prod, TEv).ProcessP(Prod, TEv).$$
$$(\overline{c_{remoteP_1}}\langle Prod, TEv\rangle \,|\overline{c_{remoteP_2}}\langle Prod, TEv\rangle \,|...|\overline{c_{remoteP_n}}\langle Prod, TEv\rangle)$$

  This means that the process receives *Prod* and *TEv* along a dedicated channel c_{localP}, and processes this information within the *ProcessP* sub-process, which determines the correspondent remote EvAs and the associated channels $c_{remoteP_i}$. *Cons* and *TEv* are sent over these channels as the event notification.

- Event notifications. The event notifications are distributed to the registered event consumers by their local EvAs. We write this as:

$$c_{remoteP}(Prod, TEv).ProcessRemoteP(Prod, TEv).$$
$$(o_1 \mapsto m_1|o_2 \mapsto m_1|...|o_n \mapsto m_n)$$

  The process receives information about the producer, respectively about the produced event along channel $c_{remoteP}$. The consumers o_1, o_2, ...,o_n and their methods m_1, m_2, ...,m_n, are identified inside the *ProcessRemoteP* sub-process. Afterwards, the process continues with the (local) method invocation of all these methods.

- Un-register a local/remote consumer. These two processes are quite similar to the processes for the registration of local/remote consumers:

$$c_{localUC}(Cons, TEv).ProcessUC(Cons, TEv).$$
$$(\overline{c_{remoteUC_1}}\langle Cons, TEv\rangle \,|\overline{c_{remoteUC_2}}\langle Cons, TEv\rangle \,|...|\overline{c_{remoteUC_n}}\langle Cons, TEv\rangle)$$

  respectively,

$$c_{remoteUC}(Cons, TEv).UnRegisterC(Cons, TEv)$$

  A separate set of channels is used for communication. *ProcessUC*, respectively *UnRegisterC* delete the information from the memory, instead of adding it.

Therefore, we may write EvA process as a parallel composition of the above-described processes, mapped onto a single machine m:

$EvA ::= \{P_1|P_2|P_3|P_4|P_5|P_6\}_m$

4.6.6 JavaSymphony Applications

A JavaSymphony application is modelled by a Pi-calculus process that runs in parallel with an associated $AppOA$. Both processes are placed onto a single computing resource:

$$\{A|AppOA_A\}_m$$

Each application has its own logic and therefore there is no generic formula to model A. However, by using JavaSymphony programming constructs (Chapter 3), a JavaSymphony application may perform a few standard operations, such as:

- **Registration with the JRS.** This means the creation of the $AppOA_A$ process, which runs in parallel with the application thread of execution.
- **Requesting VAs.** In terms of Pi-calculus, this means that a set of channels are opened for communication with remote PubOAs processes. The use of named channels is implicit in Pi-calculus, and therefore no Pi-calculus formula is necessary to model this operation.
- **Creating JS objects.** In terms of Pi-calculus, the creation of JS means sending a specific job along the channel for the chosen PubOA. The job contains information about the new object (e.g. class name, parameters). The new object is not itself a process; it represents merely information stored at the level of the PubOA.
- **Remote method invocation.** A specific job is sent to the PubOA. The job contains information about the object, its method name and parameters. The PubOA uses matching construct to identify the object and its method. The method invocation represents a sub-process of the PubOA, which is parallel composed with the rest of PubOA's sub-processes. This process has its own logic. It could interact with other PubOAs by remotely invoking methods of other JS objects, or it could interact with the local EvA by producing or consuming events.
- **Managing JS objects** (e.g. migrate or delete) is similar to the creation of JS object. A specific job is sent along the channel for the associated PubOA. No sub-process of the PubOA is created.

In summary, a JS application execution is modelled as the parallel composition between the application process $\{A|AppOA_A\}_m$ and the $JRSprocess$: $\{A|AppOA_A\}_m|JRS$. The application process A communicates with the $AppOA_A$ via restricted channels. $AppOA_A$ communicates with the $\{PubOA_A\}_i$ processes, mainly by placing jobs in their queues, which are executed as $Execute(m, p)$ processes. The rest of the components (e.g $EvAS$, NAS, $JSShell$ processes, as part of the JRS) follow their own execution logic. In the end, after interactions with JRS, the A process is reduced to 0, which means the termination of the application. Afterwards or in parallel, the remaining process (i.e. JRS) expects input on its channels from another application process.

4.7 Summary

The JavaSymphony Runtime System (JRS) is implemented as an agent based system. In this chapter, we have described the components of the JRS:

- **The JavaSymphony Administration Shell** (Section 4.2) is used to configure the available computing resources and to monitor the running JS applications.
- **The Network Agent System** (Section 4.3) comprises network agents (NAs) that monitor and manage each computing resource.
- **The Object Agent System** (Section 4.4) manages the interaction between the JavaSymphony middleware and the JavaSymphony distributed applications. For each distributed application, an AppOA is created, whereas each computing resource is associated with a PubOA. The interactions within the OAS are based on a job-processing mechanism.
- **The Event Agent System** (Section 4.5) manages the JavaSymphony distributed event mechanism. The EvAS comes as a layer between the event producer(s) and consumer(s): The producer and EvA, respectively the EvA and the consumer communicate locally, whilst remote communication exists only between distinct EvAs.

Furthermore, in Section 4.6, we have introduced a formal model based on Pi-calculus to express the behaviour of the concurrent components of the JRS.

5 Scheduling Task-based Applications in JavaSymphony

The JavaSymphony programming paradigm allows flexible implementation of a large range of distributed applications, such as meta-task applications or workflow applications. However, the developer usually has to manage the resources, build Java objects, and control the mapping of the objects onto resources. In order to improve the performance, the developer needs to use a scheduling strategy adapted to his particular application. All these issues require a significant programming effort.

On the other hand, many distributed applications follow a well-defined pattern, and therefore many of the above-mentioned programming issues could be automatized. We are particularly interested in automatic resource allocation and scheduling.

Motivated by these aspects, we have added new JavaSymphony features to support automatic scheduling of some particular types of application, namely workflow applications, which we will present in the following sections.

5.1 JavaSymphony Workflow Applications

A **workflow application** is defined as a set of one or more linked activities, which collectively realize a common goal. Information (files, messages, parameters, etc.) is passed from one participant to another for action, according to a set of procedural rules.

On the one hand, workflow applications have become quite popular in Grid community and many research and industry groups proposed standards to model and develop workflow applications and built workflow definition languages or schedulers for workflow applications [35, 36, 37, 38]. On the other hand, the workflow applications may support automatic resource discovery, allocation and scheduling.

Motivated by these aspects, we have considered to support the development of workflow applications in JavaSymphony with:

- A graphical user interface for building the structure of a workflow. The user interface also allows the specification of resource constraints and workflow activity properties and constraints.
- A simple XML-based specification language[1] for describing the workflow and its elements. A JS workflow application is automatically generated using the workflow description and the class files for its activities.
- Library support consisting of a set of classes for building workflows and customizing workflow activities.
- A specialized scheduler, which automatically finds suitable resources, maps workflow activities onto these resources, and runs the distributed computation, according to the workflow specification.

5.2 Workflow Model

A workflow consists of several interconnected computing activities. Between two computing activities there may be: (1) a control flow dependency, which means that one activity cannot start before its predecessors finished or (2) a data dependency, which means that one activity needs input data that is produced by the other.

In this section we introduce a formal representation of the workflow application and present in detail the basic elements of the workflow model.

5.2.1 Formal Representation of a Workflow Application

Each workflow application is associated in our model with a workflow graph defined by: $WF = (Nodes, CEdges, DEdges, Loops, PLoops, istate, fstate)$ where:

- $Nodes$ is the set of the vertices in the graph $Nodes = Act \cup DAct \cup Init \cup Final \cup Branches$ and comprises the vertices for all activities, dummy activities, initial states, final states and branches in the workflow.
- $CEdges$, $DEdges$, $Loops$, respectively $PLoops$ are the sets of the control links, data links, respectively loops and parallel loops of the graph. We denoted $Edges = CEdges \cup DEdges \cup Loops \cup PLoops$ the set of all edges in the associated graph.

The elements of $Nodes$ and $Edges$ sets represent the basic elements of the workflow and are explained in detail in the following section.

5.2.2 Basic Elements of the Workflow

The terminology and specifications proposed by the Workflow Management Coalition [22] are used to define the above-mentioned basic workflow elements.

[1] The XML Schema for the workflow specification language is presented in Appendix 10.4

Activities

The set $Act \subset Nodes$ includes the **activities** of the workflow, which perform its computational parts. They are represented as vertices of the associated graph. One activity has input ports, which are used to receive data from predecessors and output ports used to send data to its successors. The ports are simply identified by their index (0,1,...n-1). For simplicity, we call them input_port(0), input_port(1)..., respectively output_port(0), output_port(1)...

In addition, the activities provide workflow-relevant data that is used to control the overall execution. We assume that this data is provided in a simple format (as an integer), which may indicate the actual state of the activity (mapped, ready, cancelled, running, error, etc.), or any additional information that may be used in evaluating internal conditions.

Each activity is associated with a Java class that extends the *DAGActivity* abstract class provided by the JavaSymphony library. Instances of the associated classes are mapped onto computing resources, where they perform specific computations. Also, each activity is associated with a unique *id* within the entire workflow. Additional information that is relevant for scheduling may be associated with each activity, which includes:

- **Activity properties** like associated priority, computational load (expressed in FLOPS), number of input and output ports, and **activity constraints** like maximal execution time allowed, minimal execution time required or estimated average execution time. The scheduler uses this data for scheduling decisions.
- **Resource constraints** associated with the resources, which will be used by the corresponding activities (e.g. idle CPU time, free memory, free disk space, etc.). A resource broker uses the resource constraints within the scheduling process.
- **A performance contract** is associated with an activity and is used to detect at runtime whether the assigned resource becomes unsuitable, and, consequently, the activity needs to be migrated to a new one. The performance contract information has a similar format with the resource constraints and it is used by the resource broker as well.
- **Input parameters** are passed to the activities at runtime, directly by the enactment engine.

The code excerpt in Fig. 5.1 illustrates the use of the specification language for one activity.

Dummy Activities

The $DAct \subset Nodes$ set comprises the so-called dummy activities. These represent a restricted type of activities, which are supposed to perform evaluation of complex expressions, or setup workflow variables. It is assumed that the

```
< node id="4" type="activity" >
   <name>Multiply_4</name>
   ...
   <activity_properties>
      <class>js.test.activity.Multiply</class>
      <settings>
         <priority>5</priority>
         <input_ports>1</input_ports>
         <output_ports>1</output_ports>
         <avg_time>25.0</avg_time>
      </settings>
      </constraints>
         <constraints>
            <constraint>
               <id>benchCPU_composite_score</id>
               <operator>&gt;=</operator>
               <value>15</value>
            </constraint>
            <constraint>
               <id>cpu_idle</id>
               <operator>&gt;=</operator>
               <value>80</value>
            </constraint>
         </constraints>
         <perf_contract>
            <constraint>
               <id>cpu_idle</id>
               <operator>&gt;=</operator>
               <value>40</value>
            </constraint>
         </perf_contract>
         <parameters>
            <parameter>10</parameter>
            <parameter>256</parameter>
         </parameters>
      <parameters/>
   </activity_properties>
</node>
```

Fig. 5.1. Activity specification. Code excerpt

```
<node id="14" type="dummy_activity">
   <name>IsPositive_8</name>
   <activity_properties>
      <class>js.test.activity.IsPositive</class>
      <settings>
         <input_ports>1</input_ports>
         <output_ports>1</output_ports>
      </settings>
      <parameters>
         <parameter>Param0</parameter>
      </parameters>
      <variables>
          <set_var>
             <var_name>fLoop</var_name>
             <value>true</value>
          </set_var>
       </variables>
   </activity_properties>
</node>
```

Fig. 5.2. Dummy activity specification. Code excerpt

dummy activities require minimal computing power. Instead of being placed onto distributed computing resources, they run locally, within the scheduler.

Dummy activities may require input data from previous activities (predecessors from the control-flow dependency point of view), and allow input parameters provided by the scheduler at runtime, as the regular activities do. On the other hand, the dummy activities do not produce output data for their successors. They may provide only workflow-relevant data (i.e. data used by the scheduler to determine the future plan of execution for the entire workflow). Consequently, it is not allowed to associate constraints with dummy activities, and only a reduced set of activity properties/constraints is available for them.

The code excerpt for the specification of a dummy activity is shown in Fig. 5.2. The code shows a dummy activity, which takes an input parameter and initializes a variable *fLoop* with the value *true*.

Control Links

The elements of $CEdges$ set represent the control-links of the workflow. The control-links are used to define the **control-precedence relation** over the elements of $Nodes$ set. The **control precedence relation** represents the transitive closure of $CEdges$, and it is a partial order over $Nodes$:

$N_1 < N_p$ **iff** $\exists N_1,\ N_2 ...\ N_P \in Nodes,\ \forall i \in \{1, 2, ...p-1\},\ (N_i, N_{i+1}) \in CEdges$.

One control link $(N_1, N_2) \in CEdges$ between the two vertices associated with the activities N_1 and N_2 means that N_2 cannot be started unless N_1 is finished. In this case, we say that there is **a direct control precedence relation** between N_1 and N_2.

A path (of more than one control links) between two vertices associated with the activities N_1 and N_p implies that there is an **indirect control precedence relation** between N_1 and N_p. $N_1 < N_p$ implies also that N_p cannot be started before N_1 is finished.

Note that control links may exist between any two elements of the $Nodes$ set. We denote the predecessors, respectively the successors of workflow node as: $pred(N) = \{M \in Nodes | (M, N) \in CEdges\}$, and respectively $succ(M) = \{N \in Nodes | (M, N) \in CEdges\}$.

Each control link has a source and a target: For the control link represented as $(N_1, N_2) \in Nodes \times Nodes$, N_1 is called the source of the control link and N_2 the target. On the other hand, for each element A in $Nodes$, we call *entries* of A the control-links that have this element as target: $entries(A) = \{(B, A) \in CEdges | B \in Nodes\}$, and *exits* of A the control links that have it as source: $exits(A) = \{(A, B) \in CEdges | B \in Nodes\}$

Data Links

The **data links** define the **data-precedence relation** over the set of the activities of a workflow. A data-link between two activities means that the second activity requires as input the output data of the first. This data is transferred from one activity to another by using the remote method invocation mechanism, or alternatively by transferring a list of files from the location of the first activity (data-predecessor) to the location of the second activity (data-successor).

Note that data links are allowed only between two regular activities. In addition, the dummy activities may be only the target of a data link (are allowed to collect data from other activities, but they are not allowed to provide data to other activities). Therefore, the data-precedence relation is a partial relation, defined only over $Act \times (Act \cup DAct)$, and it is not a transitive relation.

We write $N_1 <_d N_2$ **iff** $(N_1, N_2) \in DEdges$.

Intuitively, for a workflow without loops, $N_1 <_d N_2$ requires that $N_1 < N_2$, since N_1 cannot send output data to N_2, before N_1 has been completed and has produced its output.

A data-link is associated with an output port of the source activity and an input port of the target activity, or alternatively with a list of files that need to be transferred. Additional information that is relevant for scheduling may be associated with each data link:

```
<data_link>
   <source>5</source>
   <target>4</target>
   <link_properties>
      <source_port>0</source_port>
      <target_port>0</target_port>
      <settings>
         <communication_load>15.0</communication_load>
      </settings>
      <constraints>
         <constraint>
            <id>bandwidth</id>
            <operator>&gt;=</operator>
            <value>15</value>
         </constraint>
      </constraints>
      <files>
          <file_transfer>
             <source_file>file1.src</source_file>
             <target_file>file1.trg</target_file>
          </file_transfer>
          <file_transfer>
             <source_file>file2.src</source_file>
             <target_file>file2.trg</target_file>
          </file_transfer>
       </files>
   </link_properties>
   ...
</data_link>
```

Fig. 5.3. Data Link specification. Code excerpt

- **Communication properties** like communication load (expressed in MBs) - to be used for scheduling decisions;
- **Constraints** for the physical link that are associated with the data link (e.g. bandwidth and latency) - to restrict the access to certain network resources.

Code excerpt for a data-link is shown in Fig. 5.3.

Initial and Final States

Each workflow has one entry and one exit point, which we call the **initial state**, respectively the **final state** of the workflow. These workflow elements are not associated with computation. They are used for synchronization of activities, or to mark the body of the so-called sub-workflows.

```
<node id="10" type="initial_state">
    <name>Initial state_10</name>
    ...
</node>
<node id="11" type="final_state">
    <name>Final state_11</name>
    ...
</node>
<loop>
    <source>11</source>
    <target>10</target>
    <loop_settings>
        <iterations>3</iterations>
        <termination_condition>
          <or_term>
            <and_term>
              <activity>IsPositive_8</activity>
              <operator>!=</operator>
              <value>1</value>
            </and_term>
          </or_term>
        </termination_condition>
    </loop_settings>
</loop>
```

Fig. 5.4. Initial/Final States and Loops. Code excerpt

The initial states within a workflow are the members of the $Init \subset Nodes$ set, whilst $Final \subset Nodes$ includes the final states. There is a unique $istate \in Init$, for which $preds(istate) = \emptyset$, and a unique $fstate \in Final$, for which $succs(fstate) = \emptyset$. These two elements are part of the definition of the workflow graph. The rest of the initial and final states are used to define sub-workflows of the workflow, as it will be explained in the next subsection.

There is a bijective function $final : Init \rightarrow Final$, which uniquely maps an initial state $i \in Init$ to a final state $f \in Final$. The initial state of the workflow $istate$ is mapped to $fstate$: $final(istate) = fstate$. The inverse function is $init = final^{-1} : Final \rightarrow Init$, which maps each final state to a unique initial state in $Init$.

The XML constructs for initial/final states can be seen in Fig. 5.4.

Sub-workflows

The initial and final states are used to mark sub-workflow units of a larger enclosing workflow. Each **sub-workflow** unit is delimited by a unique pair

$(i, f = final(i)) \in Init \times Final$. Note that all the activities in a (sub)workflow are control-successors of the associated initial state and control-predecessors of the associated final state.

We define a sub-workflow of a workflow as follows:

$WF' = (Nodes', CEdges', DEdges', Loops', PLoops', istate', fstate')$ is a **subworkflow of** WF **iff** WF' is a workflow with the following properties:

$$\begin{aligned}
&Nodes' \subset Nodes,\ CEdges' \subset CEdges, \text{ and } DEdges' \subset DEdges,\\
&(N_1, N_2) \in CEdges \cap (Nodes - Nodes') \times Nodes'\\
&\qquad \Rightarrow N_2 = istate' \text{ (unique entry point);}\\
&(N_1, N_2) \in CEdges \cap Nodes' \times (Nodes - Nodes')\\
&\qquad \Rightarrow N_2 = fstate' \text{ (unique exit point);}\\
&final(istate') = fstate\ .
\end{aligned}$$

The idea is that each pair $(i, f = final(i)) \in Init \times Final$ corresponds to a unique sub-workflow $WF' = (Nodes', CEdges', DEdges', Loops', PLoops', i, f)$ such as $i < N < f, \forall N \in Nodes'$.

For each sub-workflow unit, "external" control-links are not allowed. This means that it is not allowed to have control links with one end (i.e. source or target) inside a sub-workflow and with the other end outside the sub-workflow, except the entries of the initial state, respectively the exits of the final state. On the other hand, the data links may have any two distinct activities as source and target, no matter if they are outside or inside one sub-workflow.

Note that these aspects resemble the properties of a procedural programming language: *Goto-s* placed outside of a procedure that point to an instruction inside this procedure are not permitted. At the same time, *goto-s* that are placed inside a procedure and point to an instruction outside are not allowed. Nevertheless, the sub-procedures of a program may access "global" data stored in previously defined variables.

Conditional Branches

The conditional branches are the elements of the $Branches \subset Nodes$ set. The execution plan of a workflow is dynamically changed by using conditional branches. Each exit (control-link) of a conditional branch is associated with a boolean condition, as shown in Fig. 5.5. A built-in scheduler chooses at runtime to execute the branch successors (activities or sub-workflows), for which the associated conditions are evaluated as *true*. The rest of the successors (activities or sub-workflows) is omitted in the execution of the workflow and placed in "cancelled" state.

Loops

JavaSymphony extends the classical DAG(directed acyclic graph)-based workflow model by supporting workflows with (sequential) loops (not allowed in

```
<node id="15" type="branch">
   <name>Branch_15</name>
</node>
<link>
   <source>15</source>
   <target>16</target>
   <branch_condition>
      <or_term>
         <and_term>
            <activity>IsPositive_8</activity>
            <operator>==</operator>
            <value>1</value>
         </and_term>
      </or_term>
   </branch_condition>
</link>
```

Fig. 5.5. Branch and branch condition. Code excerpt

DAG-based workflows). The loops are represented by the elements of the *Loops* set from the definition of workflow associated graph.

For consistency reasons, the loops in JavaSymphony may be associated only with entire (sub)workflows units, with a single entry (initial state) and a single exit (final state) point, according to the following rules:

$$Loops \subset Final \times Init,$$
$$(f, i) \in Loops \text{ only if } f = final(i)$$

For a (sub)workflow with an associated loop, the entire sequence of activities is executed repeatedly, for a fixed number of times (for-loops), or until an associated termination condition is satisfied (until-loops). There is no explicit support for while-loops, but such a loop can be easily simulated by combining an until-loop with a conditional branch.

A fixed number of iterations and/or a termination condition can be specified for each loop, as shown in Fig. 5.4. At runtime, after all the activities within the (sub)workflow have been finished, the built-in scheduler/execution engine evaluates the termination condition, respectively checks the number of iterations already executed, and determines the future execution plan for the workflow:

- If the termination condition is fulfilled, or the number of executed iterations is equal with the fixed maximum number of iterations, the successors of the sub-workflow are enabled to run;
- If not, all the activities within the (sub)workflow are executed again.

Parallel Loops

The members of $PLoops$ set of edges are called parallel loops. The parallel loops and the regular sequential loops are both associated with sub-workflow units:

$PLoops \subset Final \times Init$,
$(f, i) \in PLoopss$ only if $f = final(i)$

However, the parallel loop models a different behaviour of the associated sub-workflow: For each parallel loop, the number of iterations n is specified, and n identical copies of the associated sub-workflow will be created and executed **in parallel**. Note that for regular loops, the associated sub-workflow is **sequentially** executed for a number of times. In addition, in contrast to the regular loops, there is no termination condition for parallel loops.

A parallel loop can be replaced with n identical copies of the associated sub-workflow, in which case the workflow graph may have a significantly more complex structure. Therefore, the parallel loop is very useful when the workflow application comprises many identical sub-parts that may run in parallel.

5.2.3 Workflow Patterns

The Workflow Management Coalition (WFMC) [22] defines terminology and standards for basic workflow constructs. A detailed overview of basic and complex workflow patterns is presented in [39]. This paper also points out to which extent current existing workflow management systems realize these patterns. It is assumed, that a workflow specification language is more expressive if it supports a larger sets of workflow patterns.

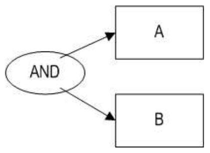
It is not our objective to support as many workflow patterns as possible. Additional complex patterns add overhead to the scheduling process. More important for us is to allow specification of scheduling relevant information. The scheduler/enactment engine should be able to easily use this data for proper scheduling decisions. On the other hand, it may be useful to investigate how some of these workflow patterns are supported by our workflow model.

Sequence

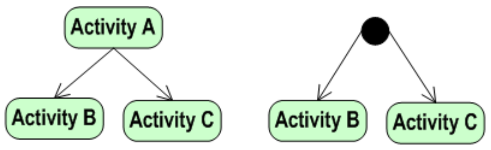
The sequence pattern is used to model consecutive steps in a workflow process. The sequence is implicitly supported via control links. Any **control-path** (e.g. a path in the associated graph defined by the control links) in a workflow defines a sequence.

AND-Split

The *AND-split* (Fig. 5.6(a)) is also known as *parallel split* or *fork*, and represents a point in the workflow execution where a single thread of control may



(a) AND-split



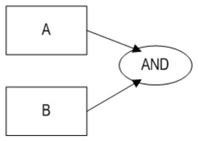
(b) JS AND-split

Fig. 5.6. AND-split pattern

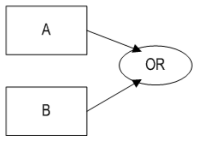
split in multiple threads. The activities in the resulting threads may run in parallel (if enough resources are available).

The existence of the *parallel split* is implicit in our model (Fig. 5.6(b)). There is no special construct for this pattern. Any activity, branch, initial or final state with several control-successors represents a *AND-split*. Its successors may run in parallel after its termination. The initial states typically represent *AND-splits*.

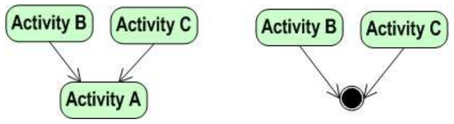
AND-Join



(a) AND-join



(b) OR-join



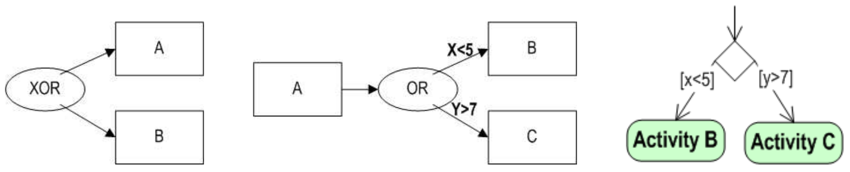
(c) JS AND-join

Fig. 5.7. AND-join and OR-join patterns.

The *AND-join* (Fig. 5.7(a)) is also known as *synchronizer* and is used to converge threads of parallel activities in one single thread of control.

The *AND-join* is also implicit in our model (Fig. 5.7(c)). There is no special construct for synchronization. Any activity with several control-predecessors represents an *AND-join*. Such an activity starts only when all its predecessors are finished. The final states typically represent *AND-joins*.

There is no rule to associate each *AND-split* with a corresponding *AND-join*, as many other workflow models do for explicit *AND-split* and *AND-join* constructs. However, a similar rule applies for initial and final states: Each initial state is uniquely associated with a corresponding final state, defining a sub-workflow unit.



(a) XOR-split (b) OR-split (c) JS OR-split

Fig. 5.8. OR-split and XOR-split patterns.

XOR-Split

It is also known as *exclusive choice* or *switch*, and represents a point in the workflow execution where one of several branches is chosen, based on a decision or on internal workflow data.

The *XOR-split* (Fig. 5.8(a)) is a particular case of *OR-split* (multi-choice) (Fig. 5.8(b)), which is explicitly represented by the conditional branch element (Fig. 5.8(c)).

Any branch (control link) that exits out of a conditional branch point is associated with a condition. A particular branch element represents a *XOR-split* only if the conditions associated with its branches are disjoint. The branch whose condition is evaluated as *true*, assuming that there is only one such a branch, is chosen for execution. The rest of them are "cancelled", which means that they will not be executed.

There is no explicit *Merge* associated with the *XOR-split*.

OR-Join

It is also known as *merge* or *asynchronous join* and represents a point where several branches come together without synchronization (Fig. 5.7(b)). It is assumed that these branches do not run in parallel; in particular that only one branch is actually executed.

In our model, there is no explicit construct for this pattern. The activities on the branches that are not chosen in a *XOR-split* are implicitly cancelled and treated as already finished. Any plain *AND-join* (e.g. an activity with several control-predecessors) that implies synchronization of a cancelled branch, acts like an *OR-join*.

OR-Split

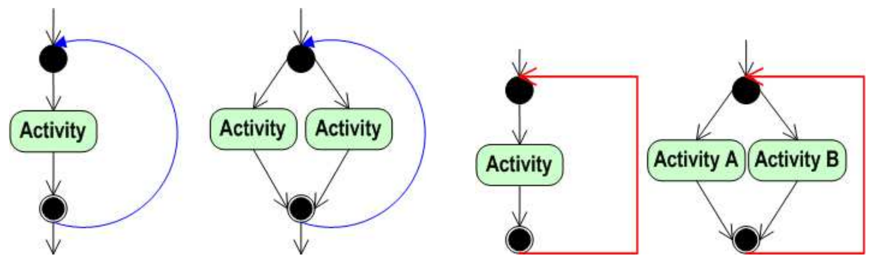
It is also known as *multi-choice* (Fig. 5.8(b)). As we stated above, the *OR-split* is explicitly represented by the conditional branch element. In case of an *OR-split*, the scheduler has to choose a subset of valid branches, in contrast to the *XOR-split* where a single branch is chosen. This is done by evaluating all branch conditions, and choosing those branches whose conditions are true.

Synchronizing Merge

It is a synchronization point associated with an *OR-split*. Multiple threads that are active due to an *OR-split* converge in such a point. The inactive threads do not need to be synchronized.

Our workflow model does not explicitly support this pattern. Any plain *AND-join* that has as predecessors both activities that are part of a cancelled branch and activities that are not part of such a branch, acts as *Synchronizing merge*. In [39], the *OR-split* and *Synchronizing Merge* are included in a list of **advanced branching and synchronization patterns**. This list includes also *Multi-merge* and *Discriminator* patterns, which are not supported by our model. We have chosen not to detail these complex patterns.

Structured Cycles vs Arbitrary Cycles



(a) Sequential loop (b) Parallel loop

Fig. 5.9. Sequential and parallel loops in JavaSymphony.

These patterns are considered structural patterns. A cycle allows the repeated execution of one or more workflow activities. The **structured cycles** can have only one entry point to the loop and one exit point from the loop and they cannot be interleaved. The structured cycles are explicitly supported in our model (Fig. 5.9(a)). The **arbitrary cycles** do not have these restrictions and are not supported by the JavaSymphony workflow model.

Patterns Involving Multiple Instance

According to [39], this category includes **multiple instances without synchronization**, **multiple instances with a priori design time knowledge**, and **multiple instances with a priori runtime knowledge**.

The JavaSymphony workflow model includes loops (Fig. 5.9(b)), which can be used to model multiple instances with a priori design time knowledge, if the number of iterations is set to a constant value (default case), or they can model multiple instances with a priori runtime knowledge, if the number of iterations is set to a workflow variable, whose value has been set by some dummy activity. Note that sequential and parallel loops are used in similar conditions: they are attached to a (sub-)workflow unit; the graphical representation is slightly different: sequential loops have a rounded form, whilst parallel loops a rectangular one.

In [39], several other categories of workflow patters are introduced (e.g. patterns involving multiple instances, stated-based patterns, cancellation patterns). These patterns have a higher complexity, and they are not commonly used in workflow applications. Due to their complexity, a scheduler will have to manage them at a higher cost. JavaSymphony does not support these patterns. We consider supporting some of them in the future, as long as the additional scheduling costs are acceptable.

5.2.4 Modelling Workflow Applications with Pi-calculus Processes

Graph-based modelling allows graphical definition of an arbitrary workflow through a few basic graph elements. The workflow graphical representation is very intuitive and can be easily managed even by a non-expert user. The workflow model discussed in the previous sections is in a direct relationship with the graphical workflow representation, which the user assembles with the help of the JavaSymphony workflow composition tool.

However, for an in-depth theoretical understanding of the model, the graphical representation is not enough. As an alternative formal model, Pi-calculus introduced in Appendix 10.3 can be used to express the dynamical behaviour of the workflow process. In contrast to graphical modelling, process algebra's are based on a textual (i.e. rather linear) description. Pi-calculus [34] is the most recent addition to the impressive collection of process algebra variants. Pi-calculus can be used to model any process, including how one workflow works, since a workflow itself is just a process. Its patterns can be constructed out of Pi-calculus primitives [40].

On the other hand, many think that Pi-calculus is at a too low-level to be used as a serious tool. However, we want to find out for ourselves if it can be useful in real world problems. Therefore, we have chosen to represent the elements of our workflow model also as Pi-calculus processes. We think that the Pi-calculus can be regarded as an internal representation of the workflow model, which allows analysis and verification.

Activities

The activities correspond to atomic Pi-calculus processes. For example, an activity A is represented as a computational process P_A, which will eventually finish. This process is associated with input and output channels, according to control- and data-precedence, as described below.

Control-links

The control-links are associated with channels, which we call control-channels. A control-link between A and B is represented as a channel c_{A-B}, such as $P_A := ...\overline{c_{A-B}}\,\langle\rangle ...$ has it as output channel, with empty output, and $P_B := ...c_{A-B}()...$ has it as input channel, with empty input.

Data-links

The data-links are associated with channels in a similar way. We call these channels data-channels. A data-link between A and B is represented as a channel d_{A-B}, such as $P_A := ...\overline{c_{A-B}}\,\langle d_A\rangle ...$ has it as output channel, with non-empty output, and $P_B := ...c_{A-B}(x)...$ has it as input channel, with non-empty input.

We denote $\widetilde{c_{inA}}$, $\widetilde{c_{outA}}$ all the input control channels, respectively all the output control-channels associated with an activity A. We use a similar notation for input/output data-channels: $\widetilde{d_{inA}}$, $\widetilde{d_{outA}}$

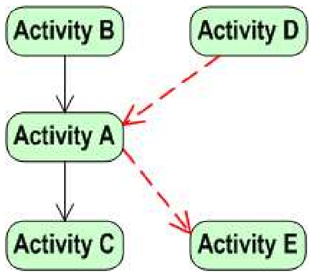


Fig. 5.10. Example of control- and data-dependency

Let's assume that A has B as control-predecessor and C as control-successor, respectively D as data-predecessor and E as data-successor (as in Fig. 5.10). Then we may write the process associated with the activity A as: $P_A := c_{B-A}().d_{D-A}(x).Comp_A.\overline{c_{A-C}}\,\langle\rangle\,.\overline{d_{A-E}}\,\langle d_A\rangle$.

According to this formula, the computation associated with the activity (written as the $Comp_A$ process) cannot start before B finishes and activates channel c_{A-B} and before D finishes and sends d_D over the channel d_{A-B}. Thereafter, the process continues as $Comp_A[d_D/x]$ (x is renamed in $Comp_A$ as the input data over the channel d_{A-B}). On the other hand, the termination of $Comp_A$ activates c_{A-C} channel and sends data d_A over d_{A-E} channel. The order in which $Comp_A$ receives input over control and data channels is irrelevant.

Dummy Activities

The formula for the dummy activities is similar. The difference lies in the fact that they do not output data and therefore they do not have output-data channels.

Initial and Final States

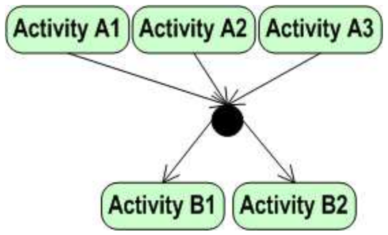


Fig. 5.11. Entries and exits of the initial state

In the formula for the initial and final states, *Comp* is replaced with 0, and input/output data-channels are not used. For example, the initial state in Fig. 5.11, can be represented as:
$P_I := c_{A1-I}().c_{A2-I}().c_{A3-I}().\overline{c_{I-B1}}\left\langle\right\rangle.\overline{c_{I-B2}}\left\langle\right\rangle$.

Sub-workflows

The sub-workflows may be represented as the parallel composition of their components. In addition, there is an initial state, respectively final state associated with the sub-workflow, which are part of the sub-workflow process:
$S := (\nu\widetilde{c})P_I|P_F|P_{A_1}|P_{A_2}|...|P_{A_n}$

At the beginning, most of the processes in the parallel composition are idle, waiting for activation over their input channels. Each sub-process continues after it receives activation signal over its control input channels and data over its data input channels, as described above. $\widetilde{c}$ denotes all the control channels, which are considered to be local for the sub-workflow. $\widetilde{c}$ includes all the channels for all the control-links within the sub-workflow and excludes the entries of the initial state I and the exits of the final states F.
$\widetilde{c} = \widetilde{c_{outI}} \cup \widetilde{c_{outA_1}} \cup \widetilde{c_{outA_2}} ... \cup \widetilde{c_{outA_n}} = \widetilde{c_{inF}} \cup \widetilde{c_{inA_1}} \cup \widetilde{c_{inA_2}} ... \cup \widetilde{c_{inA_n}}$

Note that the data links are not considered. We use the restriction for the channels in $\widetilde{c}$, to enforce the property of single entry point and single exit point of control for the sub-workflow, as expressed in the Section 5.2.1. Therefore, we may express a sub-workflow process similar to a single activity, which has the control-entries of the initial state and the control-exits of the final state:
$S = \widetilde{c_{in}}().Exec_S.\overline{\widetilde{c_{out}}}\left\langle\right\rangle$.

Note that there may be other data-channel names that are free in $Exec_S$, but there are no other free control-channel.

Branches

The branches have an input control-channel associated with the branch element entry. Each exit of a branch point is associated with a condition. We denote by $Eval(c_i)$ the evaluation of the condition associated with the i-th exit of a branch.

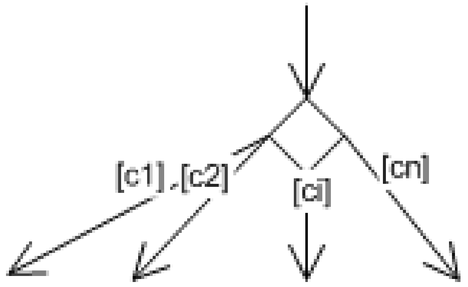


Fig. 5.12. Branch element

We use Pi-calculus *match* to write the branch process formula:

$$c_{in}().\ ([Eval(c_1) = true]\overline{c_{out1}}\langle\rangle\,|Eval(c_2) = true]\overline{c_{out2}}\langle\rangle\,|\\ ...|Eval(c_n) = true]\overline{c_{outn}}\langle\rangle)$$

where the c_{in} is the channel associated with branch element entry (a control link), and c_{out1} , c_{out2},..., c_{outn} are the channels associated with branch element exits (the actual branches) (Fig. 5.12). Note that $Eval(c_i)$ is usually a more complex process that has workflow-relevant data as input and outputs $true$, or $false$, as names on one output channel.

On the other hand, we need to model the cancellation of one activity/subworkflow, because when a branch condition evaluates to $false$, the activity/subworkflow on that branch will be cancelled. To allow the cancellation of a single activity, we extend the definition of the activity process as follows:

$$P_A := (c_{B-A}().d_{D-A}(x).Comp_A.\overline{c_{A-C}}\langle\rangle\,.\overline{d_{A-E}}\langle d_A\rangle)+\\ (cancel_A().\overline{c_{A-C}}\langle\rangle\,.\overline{d_{A-E}}\langle null\rangle)$$

We have extended the previous example of an activity process with one "cancellation component", which runs as an alternative to the main process. This is expressed in the second part of the formula. The process listens on a special input channel $cancel_A$, and simply sends a signal on all the outputs channel. This enables the control- and data-successors of the cancelled activity to run further as if this activity had finished. However, there is no data to be sent along the data-output channels.

To cancel an entire sub-workflow means that all the activities within the sub-workflow will be cancelled, as expressed in the following formula:

$$\begin{aligned} S := {} & P_I|P_F|P_{A_1}|P_{A_2}|...|P_{A_n}| \\ & \big(cancel_S().\,\big(\overline{cancel_I}\,\langle\rangle\,|\overline{cancel_F}\,\langle\rangle\,|\overline{cancel_{A_1}}\,\langle\rangle\,|..|\overline{cancel_{A_n}}\,\langle\rangle\big)\big) \end{aligned}$$

The last part of the formula models the cancellation of the sub-workflow. The sub-workflow process receives a signal on a special input channel $cancel_S$ and sends signals on the cancel channels for each inner activity. The activity processes in the first part of the sub-workflow process: $P_I|P_F|P_{A_1}|P_{A_2}|...|P_{A_n}$, are extended with the cancellation part as described above.

We can now extend the formula for conditional branches:

$$c_{in}().\,\big([Eval(c_1) = true]\overline{c_{out1}}\,\langle\rangle + [Eval(c1) = false]\overline{cancel_{out1}}\,\langle\rangle\big)\,|...$$

For each branch exit, both alternatives are modelled: either the condition evaluates to $true$, and the process associated with the subsequent activity/sub-workflow is enabled, or the condition evaluates to $false$, and the process associated with the subsequent activity/sub-workflow gets the cancel signal.

Parallel Loops

The parallel loops are attached to sub-workflows to express the fact that several identical copies of the sub-workflow are executed.

If $S = c_{in}().Exec_S.\overline{c_{out}}\,\langle\rangle$ is the formula for a sub-workflow process, and a parallel loop with n iterations is attached to this sub-workflow, we denote $S_i = c_{in_i}().Exec_{S_i}.\overline{c_{out_i}}\,\langle\rangle$, $I \in \{1...n\}$ the n identical copies of the sub-workflow process. We assume that S has a single entry c_{in}, instead of $\widetilde{c_{in}}$ and a single exit c_{out}, instead of $\widetilde{c_{out}}$ only for simplicity reason.

The Pi-calculus process associated with the loop is expressed by:

$$\begin{aligned} Par(S,n) := {} & c_{in}().\overline{c_{in_1}}\,\langle\rangle\,.....\overline{c_{in_n}}\,\langle\rangle\,| \\ & S_1|S_2|...|S_n| \\ & c_{out_1}()...c_{out_n}().\overline{c_{out}}\,\langle\rangle \end{aligned}$$

The process has three parts:

- The first part waits for input on the control-channel c_{in}. After receiving it, it signals on the input-channels associated with the n copies of the sub-workflow. This triggers the execution of the sub-workflow copies. It is not relevant if the signal on the n channels is sent sequentially as the formula shows, or in parallel.
- The second parts models the parallel execution of the n instances of the sub-workflow.

- The third part of the formula synchronizes the n copies. Signals from S_1, S_2,..., S_n are expected on their output-channels and after that a signal is sent on the c_{out}, to enable the sub-workflow successors.

Note that the channels c_{in_i} and c_{out_i} are new *internal* channels for the $Par(S,n)$ process, which means that they should be placed under the Pi-calculus *restriction* operator. For simplicity reason, we do not include the restriction in the formula.

Sequential Loops

We use the same notation to describe *identical* copies of a sub-workflow process, as for parallel loop. There are two types of sequential loops, as described in Section 5.2.1: for-loops and until-loops. Whilst **for-loops** are associated with a predetermined number of iterations, which indicates how many times the sub-workflow will be executed, the **until-loops** are associated with a termination condition, which is tested at the end of each iteration to decide whether or not a new iteration should be executed. Therefore, we model the two types of sequential loops as two distinct types of Pi-calculus processes.

The for-loops are associated with a number of iterations $iter$. We denote with $Seq_{iter}(S,n)$ the process that models a for-loop with n iterations associated to the S sub-workflow. Again, in order to simplify things, we assume that S has a single control input-channel and a single control output-channel.

We model this process as following:

$$\begin{aligned} Seq_{iter}(S,n) := \; & c_{in}().\overline{c_{in_1}}\langle\rangle \mid \\ & c_{in_1}().Exec_{S_1}.\overline{c_{in_2}}\langle\rangle \mid \\ & c_{in_2}().Exec_{S_2}.\overline{c_{in_3}}\langle\rangle \mid \\ & ... \\ & c_{in_n}().Exec_{S_n}.\overline{c_{out}}\langle\rangle \end{aligned}$$

The process is enabled upon receiving a signal on c_{in} channel. After this, it sends immediately a signal on channel c_{in_1}, which triggers the execution of the first copy of the sub-workflow. When this finishes, it sends a signal on channel c_{in_2}, instead of sending it on channel c_{out_1}. This enables the execution of the second copy of the sub-workflow. Finally, when the execution of the last copy is finished, a signal is sent on the channel c_{out}, which activates the execution of the sub-workflow successor(s).

The until-loops are associated with logical termination condition, which is evaluated at the end of each iteration. We denote by $Seq_{cond}(S, term)$ the process that models a until-loop with n iterations associated to the S sub-workflow. We model this process as:

$$\begin{aligned} Seq_{cond}(S,term) := \; & c_{in}().Exec_{s}.\overline{c_{out_1}}\langle\rangle \mid \\ & c_{out_1}().([Eval(term) = false]Seq_{cond}(S_1, term)+ \\ & [Eval(term) = true]\overline{c_{out}}\langle\rangle) \end{aligned}$$

The process is divided into three parts:

- The first part waits for input on the control-channel c_{in}. A copy of the loop is enabled to execute. At termination, it sends a signal on channel c_{out_1}, which is used to enable one of the second or third part of the process. This sub-process models the execution of one single iteration of the loop construct.
- The second part describes what happens if the termination condition is not fulfilled. The second, or alternatively the third part, is enabled upon receiving a signal on channel c_{out_1}, which was sent at the termination of the first part. If the termination condition evaluates to *false*, $Seq_{cond}(S_1, term)$ is activated, which means that a second iteration is executed.
- The third part deals with the termination of the loop. If the termination condition evaluates to *true*, it simply sends a signal on channel c_{out_1}, which enables the execution of the sub-workflow successor(s).

Again, the complexity of the process that evaluates the termination condition is hidden behind the $Eval(term)$ formula.

5.3 Building and Running Workflow Applications

To build a JavaSymphony workflow application, one has to design first the workflow graph, by using the specialized graphical user interface (Fig. 5.13). The user puts together the activities, dummy activities, initial and final states of the workflow and connects them using control links, data links, loops and parallel loops. The result is an easy-to-understand graphical representation of the workflow.

Each graphical element (vertices and edges of the graph) is associated with relevant workflow information. Some of this data is mandatory (e.g. class name for activities, activity ids, input parameters , files to be transferred, termination conditions or the number of iterations for the loops, the number of iterations for parallel loops, branch conditions, etc.). Other information is optional (e.g. performance characteristics of the computational activities or communication, resource constraints for mapping the activities or for communication network, performance contracts), but, on the other hand, it may be used by the scheduler to improve the performance of the whole application or to match the user preferences.

The workflow specification is stored as an XML file, by using the specific XML-based specification language. This file is provided to a local scheduler (build-in within the user interface or external), which performs the resource brokerage, maps the activities and enacts the whole applications.

The scheduling/enactment process consists of several operations:

- Analyzing the workflow specification for consistency.
- Resource brokerage finds the suitable computing resource for each of the workflow activities, based on the specified resource constraints.

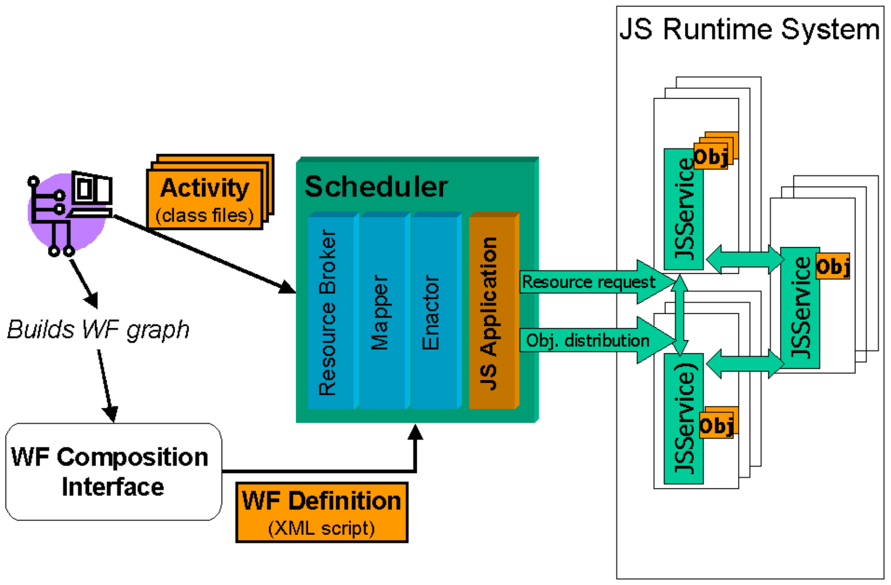


Fig. 5.13. Building and running JS workflow applications

- Building a partial execution graph. Since many of the workflow parameters are determined at runtime, only subparts of the whole workflow can be scheduled for execution at specific times.
- Scheduling the partial execution graph. A scheduling algorithm is used to determine where the activities should be placed and in which order and at what times they should be started. The scheduling algorithm uses the scheduling-relevant and performance data, as described in the workflow specification.
- Activity management. The enactment engine places and starts the activities and dummy activities, according to the scheduling algorithm. It also monitors the activity execution, and determines when the activity is finished. The execution has to preserve the control-precedence relation. The enactment engine also initiates files transfer, as defined in the workflow specification.
- Monitoring scheduling events. When the execution reaches a branch point, or a loop iteration is finished and the termination condition has to be checked, or if the performance contract for a computing resource is no longer fulfilled, a scheduling event is generated to inform the scheduler that changes of the execution plan are needed. This may imply migration of activities or recalculation of the partial execution graph.

- Application termination. The workflow application is finished when there are no more workflow activities to start, and all the activities are finished.

5.3.1 The Activity Life-cycle

A workflow application represents a collaboration of its workflow activities. The activities need to communicate with each other, but they must be able to perform their computations independently. Therefore, the implementation of the workflow activities has to obey specific rules.

Each activity is associated with a Java class that extends the *DAGActivity* abstract class, which is provided by JavaSymphony library. We identify several phases in the life-cycle of an activity, each of them being associated with a method of the *DAGActivity* abstract class:

1. **Initialization phase.** The object that performs the associated computation is created on the corresponding resource. A unique *Id* is associated with the activity. In addition, the *doInit* method of *DAGActivity* class passes to the activity the list of the data-predecessors (to collect their output) and the runtime parameters.
2. **Computation phase.** The specific computation associated with the activity is placed in the abstract *run* method. During the computation, the method *getInputData* may retrieve input data from data-predecessors. After the computation is finished, the status information or additional relevant scheduling information are collected by remotely invoking method *getStatus*. This method returns an integer, which may take predefined constant values for possible activity states (e.g. mapped, running, suspended, cancelled, error, finished, etc...), or user-defined values, relevant for the workflow scheduling. The scheduler may test these values at the time it evaluates the branch conditions or the loop termination conditions. Note that the scheduler cannot test the output data transferred from workflow activities to their data-successors.
3. **Suspended state and migration phase.** Optionally, the scheduler may decide to suspend the execution of an activity, and migrate that activity to a new computing resource. This is done by remotely invoking the method *suspend* of the *DAGActivity* class. Usually, suspending and resuming the execution are difficult to implement and application-dependent. Therefore, it is the developer's task to implement this method such that it interrupts the *run* method, saves the activity current state and sets the activity status as *suspended*. After migration, the activity is re-started by invoking the *resume* method.
4. **Output phase.** The activities, after finishing their computation, may provide output data to their data-successors. This can be done in two ways: (1) by using the JavaSymphony remote method invocation mechanism or (2) by transferring files from one location to another.

In the first case, the method *getOutputData* retrieves the output data. The data is distributed along several output ports (numbered 0,1, ...). The output port is a parameter of *getOutputData* method. Note that this method is automatically and remotely invoked by the data-predecessors' *getInputData* methods, while they are performing their computations.
In the second case, the transfer of the files, as defined in the workflow specification, is initialized by the scheduler, at the destination site. The transfer mechanism is supported by JavaSymphony and is implemented as part of the Object Agent System (see Section 4.4).

5. **Release phase.** After an activity has performed its computation and its successors have acquired all its output data, the activity is no longer needed. In this case, the resources used by the activity are released by invoking its *doRelease* method, and the activity object is deleted from the memory. Note that if an activity needs to provide data to activities in future loop iterations, the scheduler is not allowed to release it.

5.3.2 Computation and Communication

A workflow application enactment alternates computation with communication. The computation part is performed by the activities, as described above. After the scheduler has determined the mapping and the execution order of the activities, the enactment engine manages the activity state as shown in Fig. 5.14. The computation is performed as described in the previous section, after the successful mapping of the activity onto a computing resource, and after all control-predecessors of the activity have finished.

Communication is partially controlled by the enactment engine: On the one hand, this performs all the files transfers associated with the data-links; on the other hand, in the initialization phase of an activity, the enactment engine assigns the data-predecessors of an activity and the associated ports. Thereafter, the activity itself will collect data from its data-successors, as described in the previous section.

5.3.3 The Execution Plan

Based on the workflow definition, the workflow management system creates a workflow application instance as a distributed application. An execution plan is created to determine which activities are executed and in which order. Due to the presence of conditional branches or sequential loops, the execution plan has to be updated at runtime. The **workflow-relevant data** is used to control the changes in the execution plan.

Defining the Workflow-Relevant Data

There are two types of workflow-relevant data:

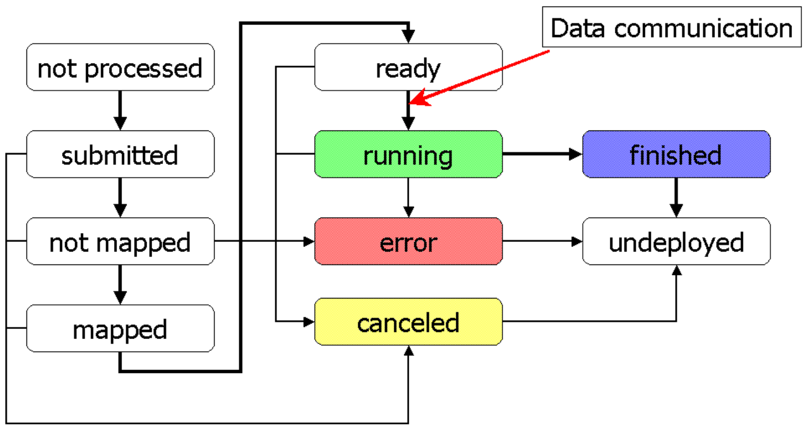


Fig. 5.14. Activity states

- **Activity state information** is determined by the enactment engine by monitoring the execution of the workflow activities. The activity state (e.g. not processed, submitted, not mapped, mapped, ready, running, suspended, error, cancelled, finished - see Fig. 5.14) can be queried and used in the evaluation of the logical conditions associated with branches or loops. Moreover, an activity or a dummy activity could set its own state to a relevant value to be used by the enactment engine.
- **Workflow variables** are defined in the workflow specification. Each sequential or parallel loop may be associated with an iteration variable, which counts the number of the iterations of the loop. The dummy activities are used to define new variables and to assign them values, based on previous defined workflow-relevant data (e.g. activity state information or other variables). The variables are used to evaluate Boolean conditions associated with branches or loops, to assign values to other variables, or to set the input parameters for an activity.

Note that the workflow activities can use data that may not be regarded as workflow-relevant (e.g. input files or data produced by other activities). The scheduler does not have a standard way to interpret this data, which may be in various formats and may have a variable data-size. In this case, an activity (most likely a dummy activity) is required to analyze the data and produce workflow-relevant data out of it (e.g. as activity state information).

Using the Workflow-Relevant Data

The workflow-relevant data is used to control the application execution plan in two workflow constructs: branches and sequential loops.

Each exit of a branch element has a branch condition associated with it. This condition is defined as a Boolean expression that uses variables and activity states, which is evaluated at runtime when the execution reaches the branch point. If the branch condition evaluates to *true*, the associated successor of the branch element may execute. Otherwise, it is placed in the cancelled state.

The sequential loops are associated with a number of iterations, or with a termination condition. In the second case, the enactment engines evaluates the termination condition at the end of each iteration. If the condition evaluates to *true*, the enactment continues with the successors of the loop exit point (a final state); otherwise it continues with a new iteration of the loop, starting with the loop entry point (an initial state).

5.3.4 An Example of Workflow Application

As we want to exemplify our workflow model, we present in this section a testing workflow application. Its associated graph (Fig. 5.15) was created by using the JS integrated graphical tool for workflow applications.

The graphical representation of the workflow is based on the UML activity diagram [41]. Each activity has a unique *id* (e.g. *Add_1*, *Multiply_4*, *Substract_10,...*) and performs associated computation.

The *IsPositive_8* activity is a **dummy activity**. It is supposed to analyze data from its predecessor and evaluate a Boolean expression.

The successor of *IsPositive_8* is a conditional branch. In this application, based on the information provided by *IsPositive_8*, one single branch will be chosen. Depending on which branch is chosen, one of the two subsequent sub-workflows is executed, and the other one is cancelled.

The workflow graph has a unique entry point (initial state), a unique exit point (final state) and it is associated with a loop, which connects the final state with the initial state. The workflow has two sub-workflows, each of them having an associated loop. Each of the sub-workflows has an initial and a final state.

Most of the graph edges represent control links (e.g. *Initial state-Add_1*, *Add_1-Multiply_4*). The data links are built along control link between activities (e.g. *Add_1-Multiply_4* or *Add_2-Substract_5*, but not *Initial state-Add_1* or *IsPositive_8-Branch*). This means that output data from one activity (e.g. *Add_1*) is transmitted as input to its data successor(s) (e.g. *Multiply_4*).

The graph shown in Fig. 5.15 is used to compute a complex mathematical expression, which has the following C-like syntax:

$$((1+4)*((2+1)-n_3)/(6-4)>0)?(8+1)*(9+4)*(10-3):(11+2)/n_{12}/n_{13}$$

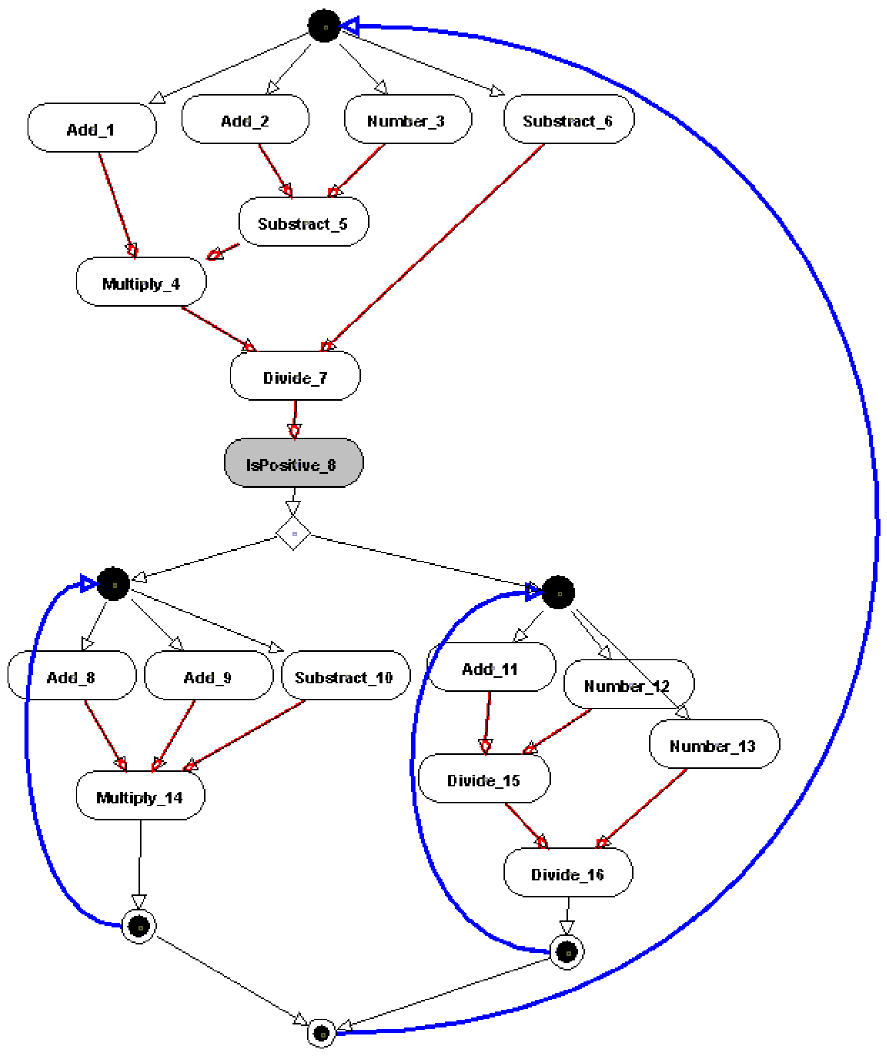


Fig. 5.15. A workflow application graph

The numeric values in the expression are passed as parameters directly to the activities. The values of n_3, n_{12} and n_{13} are generated by the activities *Number_3*, *Number_12*, respectively *Number_13*.

The activities perform only simple operations: addition, subtraction, multiplication, and division. They get input either as input parameters, or from their predecessors. The result of the expression is produced by either *Multiply_14*, or or *Divide_16* activity, depending on which branch is chosen when the execution reaches the branch after *IsPositive_8*. For testing reasons, we have artificially added loops for the entire workflow, and for the two sub-workflows. The execution of the workflow ends when all the iterations of the main loop are finished.

5.4 Scheduling Workflow Applications

5.4.1 Scheduling Workflows without Branches and Loops

We consider first the case of scheduling workflows with no loops or branches. By eliminating the conditional branches and loops, we get a workflow with a static DAG structure. The advantage is that the scheduling for DAGs of tasks has been widely studied and there are plenty of heuristics to solve this problem.

Definition 5.1. A schedule *for an application represented by* $WF = (Nodes, Edges, DEdges, Loops, PLoops, istate, fstate)$ *, with* $Loops = \emptyset$ *and* $Branches = \emptyset$ *is a function* **sched** : $Act \cup DAct \to M \times \mathbb{R}_+$.

In this definition, M is the set of computing resources and $\mathbb{R}_+$ is the set of positive real numbers. The notation $\mathbf{sched(t)} = (m_t, start_t)$ means that the task t is started on machine m_t at the time $start_t$.

For a specific schedule, we use the following notations and definitions:

- **The execution time** of one activity t on the machine m is denoted by $\mathbf{exec(t/m)}$. At this stage, we assume that the execution time is the time needed to run the task exclusively on that machine, and this value does not vary during the execution of the whole application.
- **The communication time** to send data from activity t_1 running on m_1 to activity t_2 running on m_2 is denoted by $\mathbf{comm(t_1/m_1, t_2/m_2)}$.

This data can be determined by using prediction tools and/or user estimations placed in the workflow specification for each activity, respectively data link. Note that if $t \in DAct$, we may assume that m_t is always a dedicated machine m_0 (where the scheduler is running) and we consider $exec(t/m_t)$ to be 0. We also assume that communication time for two activities running on the same machine is 0: $comm(t_1/m, t_2/m) = 0$

Definition 5.2. A well-defined schedule *of the workflow* WF *is a schedule which has the following additional properties:*

$t_1 < t_2$ *implies* $start_{t_1} + exec(t_1/m_{t_1}) \leq start_{t_2}$

$t_1 <_d t_2$ *implies* $start_{t_1} + exec(t_1/m_{t_1}) + comm(t_1/m_{t_1}, t_2/m_{t_2}) \leq start_{t_2}$

The job of a scheduler is to find a schedule for each workflow application, which optimizes a specific performance function, under certain constraints. Such functions are: makespan (execution time of the whole workflow application), total cost of the resource utilization (when the resources are associated with utilization costs) or the throughput of the entire system. The first two examples are considered to be application-level scheduling, while the last one refers to the system-level scheduling.

A few other scheduling elements are commonly used when describing a scheduling algorithm. Supposing that a scheduling algorithm provides a schedule *sched* for the pair (WF, M), then for each task t, we can define a **ready time**, a **start time** and a **completion time**.

Definition 5.3. *For a fixed schedule sched and a task* $t \in Nodes$*, for which* $sched(t) = (m_t, start_t)$

- *The* **start time of** t *is the value* $start_t$*;*
- *The* **completion time of** t *is the value* $ct(t/m_t) = start_t + exec(t/m_t)$*;*
- *The* **ready time of** t *is the value*

$$ready(t) = \max\{\max_{p<t}(ct(p/m_p)), \max_{p<_d t}(ct(p/m_p) + comm(p/m_p, t/m_t))\}$$

Definition 5.4. A control path *for a workflow* WF *without branches and loops is a series of activities* $t_1, t_2, ...t_k$*, where each pair* $(t_i, t_{i+1}) \in CEdges$*.*

Definition 5.5. *For a fixed schedule sched, we define* **the length of a control path** $P = t_0, t_1, ...t_n$ *as:*

$$length(P) = \sum_i exec(t_i/m_{t_i}) + \sum_{t_i <_d t_j} comm(t_i/m_{t_i}, t_j/m_{t_j})$$

A workflow **critical path** *is a control path with the maximum length* $\max_P(length(P))$ *from all possible control paths that start with the initial state and end with the final state.*

For a well-defined schedule *sched*, it is obvious that $t_1 < t_2$ or $t_1 <_d t_2$ implies that $ct(t_1/m_{t_1}) \leq ready(t_2)$,

and the length of a critical path CP is an inferior limit for the length of the schedule **makespan**$(sched) = \min_t(ct(t/m_t))$ (assuming that the communication and computation are not overlapped).

Note that the values of **ready time**, **start time** and **completion time**, respectively the **critical path** depend on a fixed workflow schedule. In practice, these are dynamically calculated or estimated by the scheduling algorithm, and used to gradually compute the schedule, as we will see in the next sections.

5.4.2 Scheduling Workflows with Branches and Loops

The presence of conditional branches and loops in the workflow model implies a dynamic change in the structure of the execution task graph associated with the application. Subsets of the activities composing the application may be executed repeatedly or may not be executed at all, based on data that is available only at runtime. Consequently, scheduling techniques for static DAG-based applications cannot be applied to such workflow applications.

Our strategy is to transform the application workflow into one without conditional branches or loops, and recursively find a schedule in the conditions from previous section.

First, we separate the workflow activities in two classes:

- **Unsettled activities** are the activities for which the scheduling/execution decision is taken based on data that is not (yet) available. Such activities are, for example, the activities subsequent to a conditional branch, for which the associated condition cannot be evaluated, because the parameters in the Boolean expression have not been calculated yet. Therefore, it is not sure at this point that these activities will ever be scheduled for execution.
- The rest of the activities are called **settled activities**. These are the activities that are planned for execution or have been executed at a specific time of the scheduling/execution process. All the activities for which it is sure that they will be scheduled for execution are considered **settled**.

For a workflow application, the two sets of activities are dynamically changing during execution, according to the following transformations. In the auxiliary figures, settled activities are represented as coloured vertices, whilst the unsettled activities are not coloured.

- **Parallel loop elimination** is performed before the scheduling actually starts if the number of the iterations can be statically determined. Otherwise, if the number of iterations depends on the value of workflow-relevant data (e.g. variables values), the transformation is applied upon reaching the loop entry (i.e. associated initial state).
  The parallel loop construct is used merely to reduce the complexity of the workflow structure. A workflow with the same functionality can be easily built by replacing the body of the parallel loop (i.e. the associated sub-workflow) with n identical copies, as illustrated in Fig. 5.16.
- **Branch elimination** is applied at the time the conditions associated to the conditional branches are evaluated. This transformation takes place at runtime and it is illustrated in Fig. 5.17. Note that the successors of the conditional branch have been unsettled activities before the evaluation of the condition(s), and become settled activities after that. If a branch condition evaluates to $false$, the associated branch is not executed. The transformation replaces such branches with dummy activities.
- **Transformation of for-loops.** The for-loops have a fixed number of iteration. This transformation may take place anytime during the scheduling process and it is illustrated in Fig. 5.18. For each loop iteration, clones of the activities (i.e. new activities with the same properties as the original ones) in the body of the loop and associated control/data links are added to the graph. The new cloned activities preserve the settled state, if the original activities have been settled activities before the transformation.

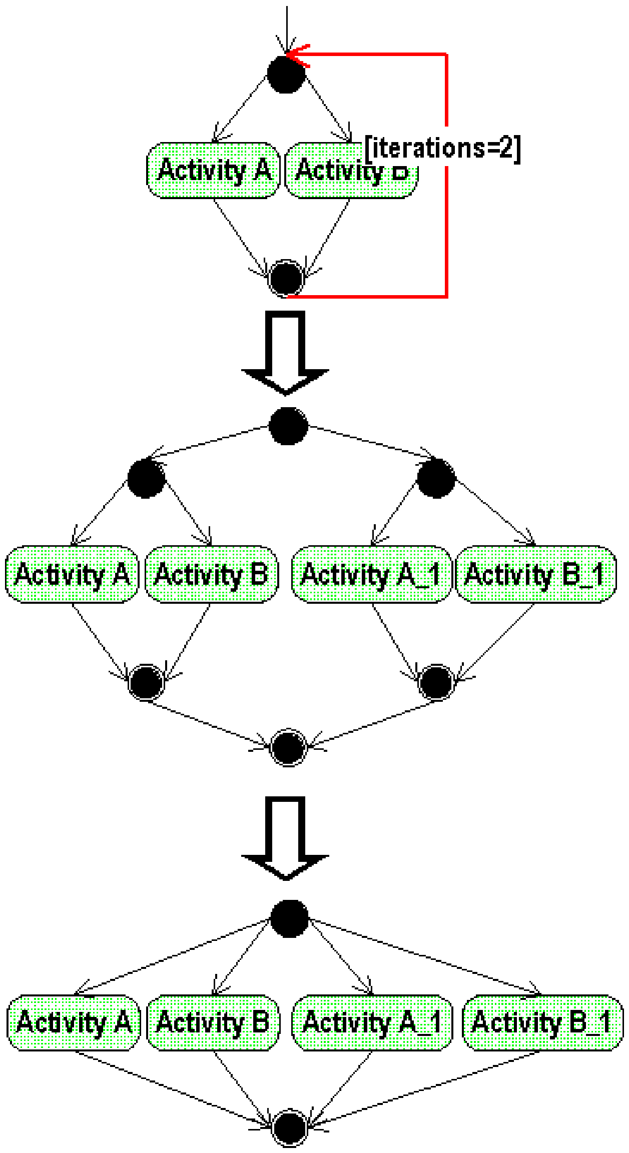


Fig. 5.16. Parallel loop elimination

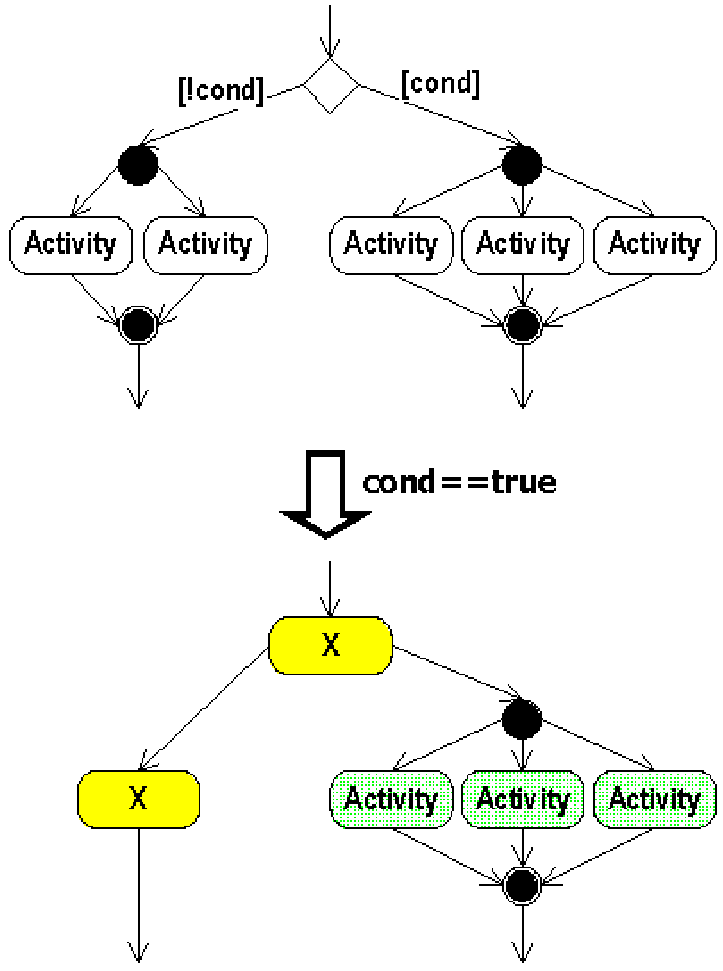


Fig. 5.17. Branch elimination

- **Transformation of until-loops.** The until-loops terminate when a specific condition is fulfilled. The evaluation of the condition can be performed only at runtime. This transformation is illustrated in Fig. 5.19. For each loop iteration, clones of the activities in the body of the loop and associated control/data links are added to the graph. The activities in the first iteration remain settled after the transformation, if they have been settled, but the clone activities in the subsequent iterations are unsettled. Any activity subsequent to the until-loop preserves its unsettled state until all the iterations of the loop are executed.
- **Elimination of initial and final states** is illustrated in Fig. 5.20. The initial and final states are simply replaced by dummy activities, not as-

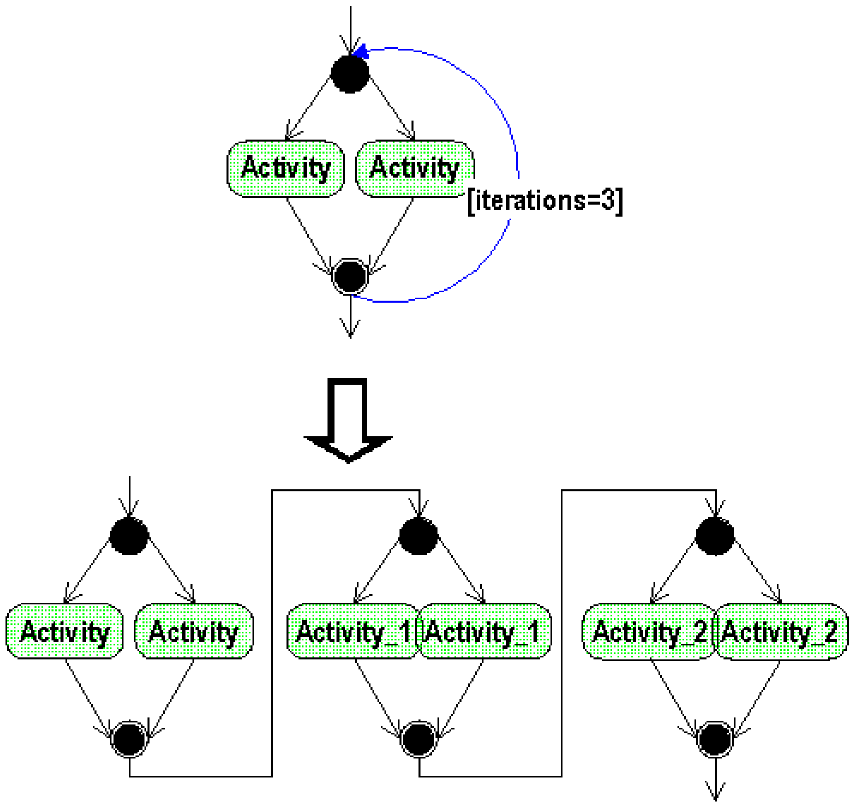


Fig. 5.18. For-loops transformation

sociated with computation. If all their (direct) predecessors are settled activities, these become settled dummy activities. The initial and the final states elements are eliminated in order to obtain a simplified graph, which has only activities, control and data-links.

Note that the branch elimination and the until-loop transformation, which implies the creation of a new conditional branch, can be performed only dynamically, at runtime. The other transformations do not depend on dynamic data, and therefore, they may be performed at design time.

We use the notation $WF \longmapsto WF_m$ to express that WF_m (modified workflow) is obtained from WF applying the above-mentioned transformations.

We iteratively build a transformed workflow as follows: Initially (prescheduling), all possible transformations, except branch elimination, are applied. The workflow application is scheduled/executed until a conditional branch is reached (i.e. all predecessors of a conditional branch finished their execution). Upon this event, the branch elimination is applied, followed by all

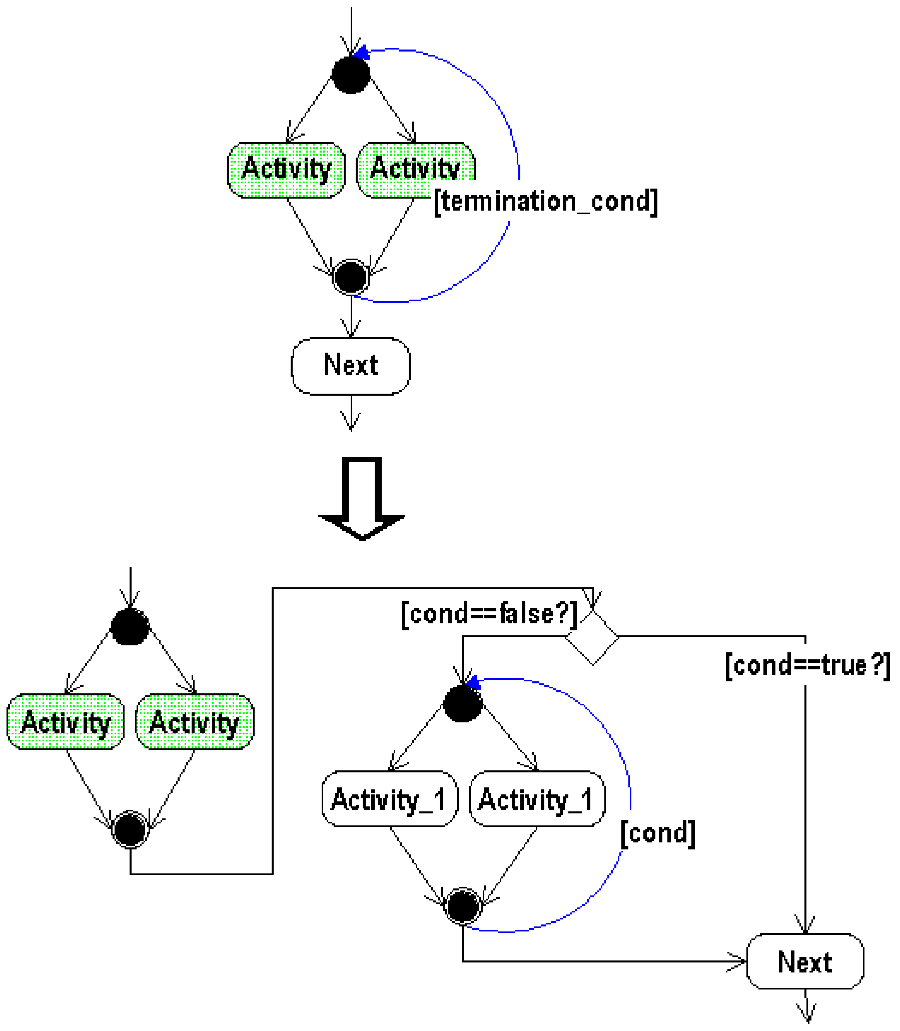


Fig. 5.19. Until-loops transformation

the other possible transformations. The sets of settled, respectively unsettled activities are recalculated after each transformation step as following.

Definition 5.6. *For $B \in Branches$ a branch node, $Next(B)$ is the set of direct successors of B, which comprises all activities directly dependent via control edges on B and all the activities of the sub-workflows directly dependent via control edges on B.*

According to this definition, $Next(B)$ includes all the activities that may be cancelled after reaching the conditional branch B. Note that the decision to

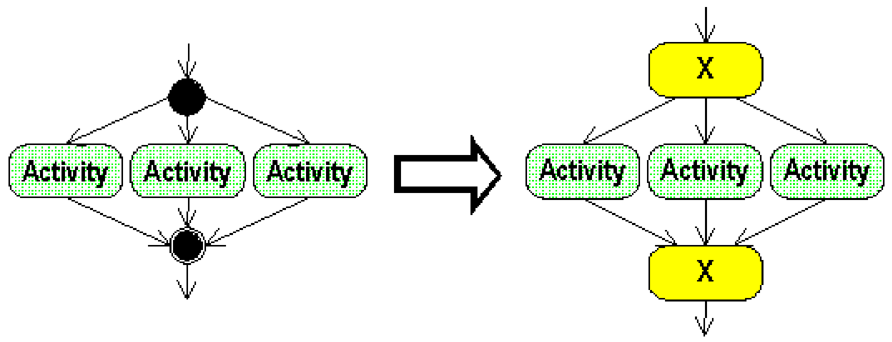


Fig. 5.20. Elimination of initial and final states

cancel or not an activity in $Next(B)$ may be taken only when the execution reaches B and all conditions associated with the subsequent branches are evaluated.

Consequently, the **set of unsettled activities** is $U(WF_m) = U_1 \cup U_2$ where $U_1 = \bigcup_{B \in Branches} Next(B)$ and $U_2 = \{N \in \mathit{Act} \cup \mathit{DAct} | \exists M \in U_1, M < N\}$

The **set of settled activities** is $S(WF_m) = \mathit{Act} \cup \mathit{DAct} - U(WF_m)$

We denote by $DAG(WF_m) = (S(WF_m), (Edges(WF_m) \cup Loops(WF_m)) \cap S(WF_m) \times S(WF_m))$, the graph which has $S(WF_m)$ as vertices and as edges all the control links, and loops from WF_m that have both the targets and sources in $S(WF_m)$.

Definition 5.7. A control path *for the workflow* $WF = (Nodes, Edges, DEdges, Loops, PLoops', istate, fstate)$ *, is a series of activities* $t_1, t_2, ...t_k$*, where each pair* (t_i, t_{i+1}) *is either a control link or a loop.*

Lemma 5.8. $DAG(WF_m)$ *is a DAG which preserves the control paths of the initial workflow.*

Proof:

- $DAG(WF_m)$ **has no loops.** According to the transformation of while loops, the body of a loop in WF_m has only unsettled activities. Therefore, the final state associated with a loop is not in $DAG(WF_m)$ and accordingly, the loop is not edge in $DAG(WF_m)$.
- $DAG(WF_m)$ **preserves the control paths of** WF. This means that for each control path $t_1, t_2, ...t_k$ of WF, with all t_i in $S(WF_m)$, there is a corresponding control path in $DAG(WF_m)$.
  First, the control edges of the initial workflow are preserved by all transformations, so if t_i,$t_{i+1} \in S(WF_m)$ and $(t_i, t_{i+1}) \in CEdges$, implies (t_i, t_{i+1}) is also edge in $DAG(WF_m)$.

On the other hand, if (t_i, t_{i+1}) is a for-loop, this means that a for-loop transformation has been applied, followed by an elimination of initial and final states. In this case, the loop is transformed into a control link between t_i and a clone of t_{i+1}, both dummy activities in WF_m.
If (t_i, t_{i+1}) is an until-loop, this means that a until-loop transformation has been applied, followed by a branch elimination and then by an elimination of initial and final states. In this case, the loop is transformed into 2 control links: (t_i, B) and B, t'_{i+1}, where B is a new branch and t'_{i+1} is a clone of t_{i+1} in WF_m and all of them are (new created) dummy activities.

Consequently, the following dynamic scheduling strategy is adopted for workflows with conditional branches and loops.

1. Apply all possible transformations on the initial workflow $WF \longmapsto WF_m$, compute $U(WF_m)$, $S(WF_m)$ and $DAG(WF_m)$. A scheduling algorithm for DAG-based workflows (no conditional branches and loops) is applied to $DAG(WF_m)$.
2. At each scheduling event, $U(WF_m)$, $S(WF_m)$ and $DAG(WF_m)$ are recalculated. Note that termination of activities may imply adding their successors to $S(WF_m)$. Changes in $DAG(WF_m)$ automatically imply scheduling/rescheduling of unfinished activities.
3. When the execution reaches a conditional branch and the scheduler evaluates the branch conditions, applies a branch elimination transformation, which is immediately followed by all other possible transformations. The result is a new WF_m, and new $U(WF_m)$, $S(WF_m)$ and $DAG(WF_m)$ are calculated. The scheduling algorithm is now applied to the new $DAG(WF_m)$.
4. The iterative scheduling/execution process finishes when all activities (in all iterations of all loops) are processed. At this point, $U(WF_m) = \emptyset$ and $S(WF_m)$ contains all the activities of WF, including the newly created activity clones (for each additional iteration of a loop), and all newly created dummy activities (for branches, initial and final states).

In the end, we have obtained a series of DAGs: WF_{m_1}, WF_{m_2}, ..., WF_{m_n} and a **well defined schedule** for each of these DAG-based workflows, according to the definitions in the previous section. The schedule obtained for WF_{m_n} is the schedule of WF, a workflow with branches and loops.

5.5 A Min-min Scheduling Algorithm for DAG-based Workflows

Min-min is a well-known static scheduling algorithm for mapping meta-tasks (large set of independent tasks) on heterogeneous computing systems. Min-min is simple; it runs fast and delivers good results. In [42], several heuristics for mapping meta-tasks are evaluated and Min-min performs well in all cases.

Only the genetic scheduling algorithm outperforms it; however, this is also due to the fact that the Min-min heuristic is used to build the initial population for the genetic algorithm. At the same time, the execution time of the genetic algorithm is significantly larger than that for Min-min.

However, Min-min is used to map tasks without dependencies, which is not the case for DAG-based workflows. Therefore, we build a scheduling algorithm, which is based on Min-min heuristics, and which can be used to schedule DAG-based workflows. The new algorithm is presented in the next section. Furthermore, in Section 5.5.2, we investigate some similar heuristics that can be easily adapted for workflow scheduling.

Similar work is done in GrADS [43]: Min-min heuristic is used to schedule DAGs of tasks onto distributed resources. However, it is not clear in which way the dependencies between the tasks influence the scheduling.

A serious limitation of the Min-min algorithm is that it does not consider the control dependencies, or data communication between the activities of the workflow. However, we think that the algorithm is still suitable for DAG-based workflow scheduling, provided that the workflow has a large number of activities that can run in parallel. We analyze this and several other limitations of the algorithm, and propose a few solutions in Section 5.5.3.

5.5.1 The Algorithm

As input for the algorithm we have the machine set $M = \{m_1, m_2 ... m_{n_M}\}$, the set of workflow tasks (activities) $Act = \{t_1, t_2 ... t_{n_{Act}}\}$, and the ETC (expected time to compute) matrix of size $n_{Act} \times n_M$. The elements of the ETC matrix describe the values of the *exec* function: $ETC(t, m) = exec(t/m)$. The algorithm produces a schedule $sched_{Min-min}$ with the properties described in Section 5.4.1.

The algorithm progressively maps workflows tasks (activities) onto the computing resources in M set. At each step of the algorithm, the tasks are divided into two sets: the set of already mapped tasks $MAct$ and the set of tasks which are not yet mapped $Act - MAct$. We use the following notations:

- $avail(m)$ denotes the availability of the machine m and represents the time when m becomes idle, after the termination of all tasks that have already been mapped onto m.
- $ready(t)$ denotes the time when "the last" control predecessors of the task t finishes. This is the moment when t is allowed to start its execution.
- $ct(t/m)$ denotes the completion time of the task t on the machine m (determined by prediction and/or user specification).

The value of $avail(m)$ is computed as $avail(m) = start_t + exec(t/m)$, where t is the last mapped task on the machine m. Note that for tasks with no dependencies, this is the sum of the execution times for the tasks mapped on this machine, but due to the dependencies in the DAG, idle times on the machine may occur between tasks.

The $ready(t)$ and $ct(t/m)$ values correspond to the definitions in Section 5.4.1. However, in this case, they are not associated to a fixed schedule, but are calculated recursively according to the scheduling strategy. The value of $ready(t)$ is computed as:

$$\max\{ct(p/m_p)|p \in pred(t), sched(p) = (m_p, start_p)\}$$

This value depends on the completion times for the control-predecessors of the task t. We assumed that they have already been mapped.

The completion time for already mapped tasks is calculated as:

$$ct(t/m) = start_t + exec(t/m), t \in MAct$$

However, the scheduling algorithm actually needs the values of $ct(t/m)$ for the tasks that are not mapped (yet). For these tasks, $ct(t/m)$ is computed according to the following formula:

$$ct(t/m) = exec(t/m) + \max(avail(m), ready(t))$$

We assume again that all the predecessors in $pred(t)$ have already been scheduled, otherwise $sched(p)$ in the definition of $ready(t)$ is not defined and $ct(t/m)$ cannot be calculated. Therefore we split the $Act - MAct$ in two subsets:

- $RAct$ denotes the set of tasks for which $ct(t/m)$ can be computed. We call them **ready tasks**. All the predecessors have already been scheduled for these tasks.
- $NAct$ denotes the set of tasks for which $ct(t/m)$ cannot be computed. We call these tasks **blocked tasks**. For these tasks, at least one of their direct predecessors has not been scheduled.

Note that in case of tasks with no dependencies, $ct(t/m) = exec(t/m) + avail(m), \forall t \in Act - MAct$ and $NAct = \emptyset$.

We use these notations to build the Min-min scheduling algorithm for DAG-based workflows, as presented in Fig. 5.21.

The algorithm attempts to minimize the **makespan**, which is computed as the value of $\max\{ct(t/m)|t \in Act, m \in M\}$. Initially, the set of mapped tasks $MAct$ is empty, the set of ready tasks is filled with the activities that do not have predecessors and the rest of the activities are placed in $NAct$. At each scheduling step, the algorithm chooses a single activity and maps it on a selected machine, according to the Min-min scheduling strategy. The availability of the chosen machine $avail(m)$ and the activity sets $MAct$, $NAct$, $Ract$ are updated.

The original Min-min heuristic is comprised in step 2 of the new algorithm The novelty of our method is the management of $MAct$, $RAct$ and $NAct$ activity sets. The Min-min heuristic is applied only to the tasks in $RAct$, whose predecessors have already been processed. $MAct$ comprises the processed (i.e. scheduled) tasks, whilst $NAct$ contains the tasks which cannot be yet analyzed, since they still have predecessors that have not been processed.

```
1. Initially $MAct = \emptyset$, $RAct$ is the set of the activities with no predecessor,
   $NAct = Act - RAct$.
2. Min-min step for the tasks in $RAct$
   • For each $t \in RAct$ compute $mct(t) = \min_{t \in M} ct(t/m)$.
   • Choose $t$ which gives the overall minimum and $m_t$ the machine for
     which this minimum is obtained.
   • Choose $start_t = \max(avail(m_t), ready(t))$, and $sched(t) =
     (start_t, m_t)$
3. Update $MAct = MAct \cup \{t\}$, $RAct = RAct - \{t\}$, and $avail(m_t) = ct(t)$.
4. For each $s \in succ(t)$, if $pred(s) \subset MAct$ add $s$ to $RAct$ and remove it
   from $NAct$.
5. Repeat from step 2 until $RAct = NAct = \emptyset$.
```

Fig. 5.21. Min-min scheduling algorithm for DAG-based workflows

5.5.2 Applying the Algorithm to Similar Heuristics

The algorithm 5.21 uses the Min-min strategy to choose the task t in step 2. However, the choice is restricted to the tasks in the set $RAct$, and steps for updating $Ract$, $MAct$ and $NAct$ are added to the original algorithm. These steps represent the key part of the algorithm, which allow the scheduling of complex task graphs instead of simple meta-tasks.

Using the structure of the above algorithm, we can easily transform some other well-known heuristics for mapping meta-tasks, and use them to schedule DAG-based workflows.

For this purpose, we have to modify step 2 of the algorithm according to the strategy of the static heuristic scheduling algorithm. Some of the heuristics presented in [42] may be used for this purpose:

- **OLB: Opportunistic Load Balancing.** The static version of this algorithm assigns each task in arbitrary order, to the next available machine. In the modified algorithm, step 2 chooses an arbitrary task $t \in RAct$
- **UDA: User-Directed Assignment** (also known as LBA - Limited Best Assignment) assigns each task in arbitrary order to the machine with the best expected execution time. For scheduling DAG-based workflows, in step 2 the task t is chosen from $RAct$. This algorithm is likely to choose only the best machines and ignore the poor ones.
- **Max-min** is similar to Min-min algorithm. The difference is that the task t is chosen such that it maximizes $mct(t)$ (instead minimizing it for Min-min). This algorithm favours long-running tasks, which are delayed in the Min-min algorithm.
- **Sufferage algorithm** [44] chooses the task t with the highest sufferage value(i.e. the difference between its best and second best completion time).

The heuristics have various complexities and each one may outperform the others for specific inputs. One user may experiment and determine which one suits better the needs of a specific meta-task or workflow application.

A few other static mapping heuristics for meta-tasks are presented in [42]. For several reasons, we cannot use the same scheduling strategy for these algorithms:

- **Genetic algorithms (GA), Simulated Annealing (SA) and Genetic Simulated Annealing (GSA)** use a representation of the solution as a chromosome and try to find the chromosome with the best fitness value (the makespan). The problem is that the transformations applied by these algorithms to the chromosomes do not preserve the precedence relation, and the result is no longer a feasible scheduling solution.
- **Tabu search and A^* heuristic** perform extensive search in large solution spaces and therefore are not compatible with the single choice of a task in step 2 of the algorithm. Similar algorithms of higher complexity need to be built for DAG-based workflow scheduling. On the other hand, [42] shows that these algorithms do not outperform the Min-min algorithm in most cases and require significantly more time for schedule computation.

5.5.3 Limitations of the Algorithm

The algorithm 5.21 is simple, runs fast and it is easy to understand. However, it does not consider some aspects of the scheduling problem, which may affect the quality of the schedule. We analyze some of these aspects in this section. We also propose a few strategies to improve it and to adapt it to our workflow model.

Idle Times Between Tasks

The classical Min-min heuristic favours the short-running tasks, whereas the long-running tasks are delayed. Min-min scheduling may yield load imbalance due to a small number of tasks with long execution tasks and unfairness for the longer tasks.

The original Min-min heuristic has been designed for scheduling meta-tasks, which comprise multiple independent tasks. In this case, the machines become idle only when there are no more tasks to be scheduled. On the other hand, scheduling tasks with control dependencies may produce idle times between tasks mapped on the same machine. Note that the scheduler chooses $start_t = \max(avail(m_t), ready(t))$.

If $avail(m_t) < ready(t)$, the interval $[avail(m_t), ready(t)]$ is idle time on the machine m_t. In the opposite case, if $avail(m_t) > ready(t)$, the task t is delayed because there is no available machine at the right moment.

The schedule can be improved by adopting a back-filling strategy, which implies accounting the "empty slots" of the machines (e.g. idle time intervals between tasks) and trying to fit the tasks into these empty slots. A similar strategy is used in ISH (Insertion Scheduling Heuristic) [45, 46] or HEFT (Heterogeneous Earliest Finish Time) [47]. The main drawbacks of this approach is that placing a ready task into an empty slot complicates the calculation of the values for task completion time and machine availability, and increases the complexity of the algorithm.

Scheduling Dummy Activities

The algorithm does not consider the existence of dummy activities. Our workflow model uses dummy activities as activities that do not perform significant computations. On the other hand, the transformations of a workflow with loops and conditional branches introduce new dummy activities in the associated graph.

Managing regular and dummy activities in different ways may improve the performance of the schedule. The dummy activities do not use resources and we assume that their execution time can be approximated with 0. We further assume that the dummy activities are mapped onto a dedicated machine m_0, probably the same that the scheduler uses. Therefore, the schedule for a dummy activity d is $sched(d) = (m_0, ready(d))$ and $ct(d) = ready(d)$.

Note that $sched(d)$ does not depend on the availability of the machine, but depends on the completion time of its predecessors. However, the dummy activities may have control-predecessors and control-successors, and therefore they influence the scheduling. The successors of one dummy activity d are considered in step 2 of the algorithm only after d was scheduled. If step 2 of the algorithm process a dummy activity d as soon as possible (even if non-dummy tasks with smaller completion time are available), the $RAct$ set is enlarged with the successors of d without updating the values of $avail(m), \forall m \in M$. With a larger $RAct$ set, the makespan of the schedule is more likely to be improved. In conclusion, the algorithm should process the dummy activities prior to the others.

Scheduling Workflows with Loops and Conditional Branches

The algorithm does not consider loops and branches. In fact, as presented above, it manages a fixed DAG associated with the workflow. For workflows with loops and conditional branches we apply an algorithm based on the transformations in Section 5.4.2. The algorithm dynamically schedules and executes a workflow with loops and conditional branches.

Reaching a conditional branch or the end of until-loops causes subsequent runtime transformations of the workflow. Other scheduling events includes termination of activities (successful or with error), performance contract violation, and user intervention:

```
1. Apply all possible transformations as defined in 5.4.2.
2. (Re)compute the sets of settled activities $S(WF_m)$, unsettled activities
   $U(WF_m)$ and determine $DAG(WF_m)$.
3. Apply Algorithm 5.21 to the subgraph $DAG(WF_m)$.
4. Go to step 1 whenever a scheduling event occurs.
```

Fig. 5.22. Dynamic scheduling algorithm for workflows

- On termination of activities, new scheduling data may be available so that the conditions associated to the conditional branches may be evaluated. This implies adding new activities to $S(WF_m)$ and enlarging $DAG(WF_m)$. Activity termination that occurs either sooner, or later than estimated implies new estimations for subsequent tasks.
- Performance contract violations usually imply new estimations for execution time of a task and do not require the recalculation in step 1.
- User intervention means manual (and unexpected) modifications of the workflow during the scheduling/execution process. The user may decide to cancel the execution of one activity, parts of the workflow, or even the whole workflow application. The $S(WF_m)$ and $U(WF_m)$ are modified by removing the cancelled tasks.

Resource Constraints

The specification language allows the user to specify constraints for the resources, which may be used by a task. However, the scheduling algorithm does not analyze or use in any other way the constraints associated with the tasks.

A separate resource broker has to do that. For each workflow activity, the set of potentially computing resources is built by eliminating the resources that do not fulfil the specific requirements. The resource broker may associate the unsuitable resources for a task with very large estimated computing times (ETC elements), so that the scheduling algorithm will never choose such a resource. If we assume that the relevant resource properties do not change, the resource broker can produce its results before the scheduling process. On the other hand, if resource behaviour changes during runtime and affects resource suitability for specific tasks, the resource broker needs to run in parallel with scheduling/enactment process.

Communication Overhead

The algorithm ignores the communication costs. When large amounts of data need to be sent from one activity to another, the communication overhead can be significant, thus affecting the overall performance of the workflow applications. We propose two methods to deal with this problem:

- Include the communication time in the estimation of completion time. For any task t that has data-predecessors, the algorithm estimates

$$comm(p/m_p, t/m), \forall p \in dpred(t), \forall m \in M$$

Depending on the implementation of data transfer, the communication overhead can be included in the estimation of completion time in several ways. On the one hand, if the data transfer is initiated at destination site, by the activity which is ready, the communication overhead may be estimated either as (1) the sum of these values: $\sum_{p \in dpred(t)} comm(p/m_p, t/m)$ if the transfer is done sequentially, one predecessor at a time; or as (2) the maximum of the communication times: $\max_{p \in dpred(t)} comm(p/m_p, t/m)$, if the data transfers can be overlapped.

The overhead estimated value is then included in the estimation of completion time for t. For the tasks that are not yet mapped, the $ct(t/m)$ is computed according to the new formula:

$$ct(t/m) = exec(t/m) + \max(avail(m), ready(t)) + \bigoplus_{p \in dpred(t)} comm(p/m_p, t/m)$$

The operator $\bigoplus$ stands either for sum or for maximum.

On the other hand, the communication may be initiated at the predecessor site and may be overlapped. In this case, the changes are reflected in the formula for $ready(t)$ according to the definition of **ready time** for a task in Section 5.4.1:

$$ready(t) = \max\{\max_{p \in pred(t)}(ct(p/m_p)), \max_{p \in dpred(t)}(ct(p/m_p) + comm(p/m_p, t/m_t))\}$$

Accordingly, the $ct(t/m)$ formula does not explicitly include the communication overhead:

$$ct(t/m) = exec(t/m) + \max(avail(m), ready(t))$$

- Consider communication as activities of the graph. Any data link is transformed into an activity of the associated graph. We call these activities transfer activities/tasks. The network links are added to the set of resources. The computation tasks are mapped only on computing resources, while transfer tasks are mapped only on network link resources.
  Additional constraints must be fulfilled: Any transfer task has a single predecessor - the computation task that produces the data; and a single successor - the computation that consumes the data. At the same time, the transfer task has to be mapped onto the network link resource that connects the two computing resources where its predecessor, respectively its successor are mapped. Note that the resource broker needs to analyze the suitability of the resources at runtime. These aspects show us the main drawback of this method: a significant higher complexity of the scheduling.

Accuracy of the Estimations

Most scheduling algorithms assume that the estimations (e.g. for ETC elements, communication time, etc) are accurate enough and take proper scheduling decisions based on these estimations. Our algorithm (Fig. 5.21) is no exception to this rule. JavaSymphony offers support for estimating the execution time of the tasks. The computing power of the resources (FLOPS/sec) is estimated based on a build-in benchmark system. In addition, the JavaSymphony Runtime System continuously monitors the dynamic parameters of the resources (e.g. CPU load, free memory, etc.). The computing requirements of a task (FLOPS) are specified in the workflow definition script. Based on this data, in a simple performance model, an estimation of the execution time for each (task,resource)-pair can be calculated as the quotient between the computing requirements of the task and the computing power of the resource. Performance prediction tools [48] may use more complex performance models, which utilize dynamic information like CPU load, or free memory to predict more accurate execution times. However, this is beyond the purpose of this work.

A basic assumption of the scheduling algorithm is that the corresponding estimation is accurate. On the other hand, the algorithm in Fig. 5.22 proposes a dynamic scheduling approach, which combines the estimated values with runtime information to compute a schedule for a workflow applications. Therefore, the estimations may be dynamically updated at runtime, whenever a scheduling event occurs. A resource broker, running in parallel with the scheduler/enactment engine has the role to update this information. More details about the resource broker are presented in Section 6.

5.6 HEFT Algorithm for Workflows

As we have seen in the previous section, the scheduling algorithms for metatasks do not consider dependencies between the tasks, which introduces a series of limitations. Managing these dependencies requires additional processing. On the other hand, the general DAG scheduling problem has been extensively studied and many research efforts have proposed heuristics to solve this problem, both for homogeneous and for heterogeneous domains [46, 49, 45, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 47, 70, 71, 72, 73, 74].

A significant number of the proposed heuristics are based on the *list scheduling* technique. The basic idea is to assign priorities to the workflow activities, and to place the activities in a list in descending order of priorities. The activities with higher priority are examined for scheduling before those with a lower priority. The list scheduling algorithms are known to perform well at relatively low cost. Therefore, we have decided to use a similar technique for workflow scheduling.

There is a large number of list scheduling algorithms for the DAG scheduling problem. However, all of them have a static approach, which computes the schedule at compile time and do not address the problem of scheduling conditional branches and loops. We have chosen HEFT (Heterogeneous Earliest Finish Time) [47], which is considered to be an important representative for list scheduling algorithms for heterogeneous systems [74, 73]. We apply the technique described in 5.4.2 to create a new workflow scheduling algorithm based on HEFT.

5.6.1 Preliminaries

The Heterogeneous-Earliest-Finish-Time (HEFT) algorithm is a DAG scheduling algorithm that supports a bounded number of heterogeneous processing elements. According to list scheduling technique, the algorithm first computes priorities for each of the workflow activities, and then processes the activities in descending order of their priorities.

We used the same notations as for min-min based scheduling algorithm. As input for the algorithm we have the machine set $M = \{1, 2...n_M\}$, the set of workflow tasks (activities) $Act = \{1, 2...n_{Act}\}$, and the ETC (expected time to compute) matrix, whose elements are $exec(t/m)$ values for each pair $(t, m) \in Act \times M$. Additionally, the algorithms consider also the communication costs, defined as:

$$comm(t_1/m_1, t_2/m_2) = data(t_1, t_2)/rate(m_1, m_2)$$

$Data = (data(t_1, t_2))$ is a matrix $n_{Act} \times n_{Act}$ for the sizes (in bytes) of the data transfers between the activities. The transfer rates between machines/processors are denoted by $rate(m_1, m_2)$, and are stored in a $n_M \times n_M$ matrix. When $m_1 = m_2$ (i.e. both activities are mapped onto the same machine), the communication cost $comm(t_1/m, t_2/m)$ becomes 0.

The algorithm uses **the earliest start time** $st(t/m)$ and the **earliest completion time** $ct(t/m)$ of an activity t on machine m, defined as:

$$\begin{aligned} ready(t/m) &= \max\nolimits_{p \in pred(t)} \{comm(p/m_p, t/m) + ct(p/m_p)\} \\ st(t/m) &= \max(avail(m), ready(t/m)) \\ ct(t/m) &= st(t/m) + exec(t/m) \end{aligned}$$

In the first phase, HEFT algorithm computes so-called *upward rank* of an activity t, used as the task priority:

$$rank_u(t) = \overline{exec(t)} + \max_{s \in succ(t)} (\overline{comm(t, s)} + rank_{u(s)})$$

where $succ(t)$ is the set of immediate successors of task t and $\overline{exec(t)}$ and $\overline{comm(t, s)}$ are *the average execution cost* of task t, respectively *the average communication cost* of edge (t, s), defined as:

$$\overline{exec(t)} = \frac{1}{|M|} \sum_{m \in M} exec(t/m)$$

$$\overline{comm(t_1, t_2)} = data(t_1, t_2)/\overline{rate}$$

$\overline{rate}$ is the average transfer rate between the machines in the domain.

The *upward rank* is computed recursively, starting from the exit node(s). It can be clearly seen that each task is ranked higher than its successors. In the second phase, the tasks are processed in descending order of their rank. For each task, the machine which gives the best completion time $ct(t/m)$ is chosen. At any moment, all the predecessors of the current activity t have been processed, since they have higher ranks. Therefore $ct(t/m)$ can be computed.

5.6.2 HEFT-based Workflow Scheduling Algorithm

```
1. Apply all possible transformations to produce $WF_m$, as described in 5.4.2.
2. Perform next steps whenever a scheduling event occurs
3. begin
4.     (Re)Compute $U(WF_m)$, $S(WF_m)$ and $DAG(WF_m)$ as in 5.4.2
5.     Eliminate finished tasks from $DAG(WF_m)$ and apply the next step
6.     Apply HEFT strategy
   • Determine exit nodes of the reduced $DAG(WF_m)$
   • Recursively compute $rank_u(t)$, starting from the exit nodes.
   • Build a list of activities, sorted by descending order of $rank_u$ values.
   • while the list is not empty
   • begin
   •     Remove $t$, the first element from the list
   •     Compute $ct(t/m)$ for each $m$ and assign $t$ to $m_t$ that minimizes it.
   • end
7.     Start the activities on the assigned machines, in the ascending order of
   $st(t/m)$ values; Each activity is started only after all its predecessors have
   finished
8. end
```

Fig. 5.23. HEFT-based algorithm for scheduling workflows

We use the technique introduced in 5.4.2, the HEFT strategy and the notations described above to create a new algorithm for scheduling workflows.

The algorithm is shown in Fig. 5.23. Note that the algorithm recursively computes a partial DAG and a partial schedule. The schedule is dynamically updated, if necessary, at runtime, on scheduling events (e.g. termination of activities - successful or with error, performance contract violation, or user intervention). The termination of activities and evaluation of Boolean expression associated with conditional branch or loops are mainly responsible for the recalculation of $DAG(WF_m)$ in step 4.

The algorithm finishes when all the activities in $DAG(WF_m)$ have finished and no other scheduling event occurs.

5.7 The Schedule Objective Function

The purpose of a scheduler is to optimize a specific function associated with the workflow, under the workflow constraints, as defined by the workflow dependencies and the resource constraints. We call such a function **the objective function** of the schedule. For a workflow WF, a set of resources M and a schedule function *sched* for the workflow WF and the resource set M, we denote the objective function with $F(WF, M, sched)$. Since WF and M are already included in the definition of the function *sched*, we may omit them and simply write: $F(sched)$. We assume that the purpose of a scheduler is to find $sched_{opt}$ such as:

$$F(WF, M, sched_{opt}) = \min_{sched} F(WF, M, sched)$$

We denote with $S_{opt}(WF, M)$ the function that associates to a workflow WF and a set of resources M, its optimal schedule $sched_{opt}$. Under these conditions:

$$F(S_{opt}(WF, M)) = \min_{sched} F_{sched}(WF, M)$$

Note that a specific scheduling algorithm Alg provides an "approximation" of S_{opt}, which we denote by $S_{Alg}(WF, M)$, and which associates to each pair (WF, M), the schedule obtained by applying the algorithm Alg to WF and M. The following definitions explain what does it mean that a scheduling algorithm provides **an approximation** of the optimal schedule.

Definition 5.9. *For a workflow WF and a set of resources M,* **the absolute error** *of a scheduling algorithm Alg is defined as the value of:*

$$\Delta_{Alg}(WF, M) = F(S_{Alg}(WF, M)) - F(S_{opt}(WF, M))$$

Definition 5.10. *For a workflow WF and a set of resources M,* **the relative error** *of a scheduling algorithm Alg is defined as the value of:*

$$\delta_{Alg}(WF, M) = \frac{\Delta_{Alg}(WF, M)}{F(S_{opt}(WF, M))}$$

Definition 5.11. *For a workflow WF and a set of resources M, we say that the algorithm provides an* ϵ-**approximation** *of the optimal schedule S_{opt}, or that $S_{Alg}(WF, M)$* ϵ-**approximates** *$S_{opt}(WF, M)$, if $|\delta_{Alg}(WF, M)| < \epsilon$.*

We write this property as: $S_{Alg}(WF, M) \simeq_{\epsilon} S_{opt}(WF, M)$,

Definition 5.12. *We say that* $S_{Alg}(WF, M)$ **approximates** $S_{opt}(WF, M)$, *and we write this as:* $S_{Alg}(WF, M) \simeq S_{opt}(WF, M)$, *if there is an acceptable small* ϵ *such as* $S_{Alg}(WF, M) \simeq S_{opt}(WF, M)$.

Moreover, we say that S_{Alg} **approximates** S_{opt}, *if there is an acceptable small* ϵ *such as* $S_{Alg}(WF, M) \simeq_{\epsilon} S_{opt}(WF, M)$, *for any* WF *and* M.

In practice, the calculation of an optimal schedule for a workflow is not feasible, since the scheduling problem is NP-complete [21] in most of the cases. Therefore, the absolute and relative scheduling error are usually not calculated and the ϵ-approximation has only a theoretical use.

In the previous sections, we have assumed that the objective function is the makespan of a schedule (i.e. the time it takes to execute the entire workflow application). However, the users may be interested in optimizing some other metrics associated with a workflow, e.g. the total cost of the resources, if the resource usage is associated with a price, overall network communication, or resource utilization (e.g. CPU or memory utilization). In this section, we investigate a few alternative objective functions and some of their properties.

5.7.1 The Execution Time as Objective Function

The purpose of the scheduling algorithms described above is to minimize the execution time. This means to find a $sched_{opt}$ scheduling function which minimize the $Makespan(sched) = \max_{t \in Nodes}(ct(t)$. Since this is a NP-complete problem, the heuristics provide a $sched^*$ "almost optimal", for which the value of $Makespan(sched^*)$ is "as close as possible" to $Makespan(sched_{opt})$. Note that there is no clear definition for "almost optimal" or "as close as possible", especially if the optimum calculation is not feasible due to large problem size (i.e. machines number and activities number). Nevertheless, we can say that a scheduler outperforms another one if it provides a smaller schedule makespan value, which implies that it has a smaller (both absolute and relative) scheduling error. In other words, for each workflow application, the objective function gives us a **order relation** over the **set of possible schedules** and an optimal schedule is a **minimum** in this set.

Let's analyze some properties of the objective function.

It is obvious that $Makespan(S_{opt}(WF, M)) \leq Makespan(S_{Alg}(WF, M))$, for any heuristic algorithm Alg.

We identify two important properties of the $F(S_{opt})$ function, which maps each workflow and each set of resources to the (theoretical) optimal value of the schedule objective function.

Monotony. Monotonic functions map ordered sets into other ordered sets, and preserve the order relation. Since we do not have an order relation over the set of workflows, we cannot mathematically define this property. Instead, we intuitively observe that if one workflow graph WF_1 is part of a larger enclosing workflow graph WF_2, then $F(S_{opt}(WF_1, M)) \leq F(S_{opt}(WF_2, M))$.

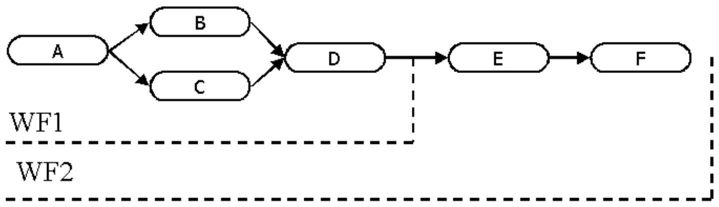


Fig. 5.24. Monotony property for workflows

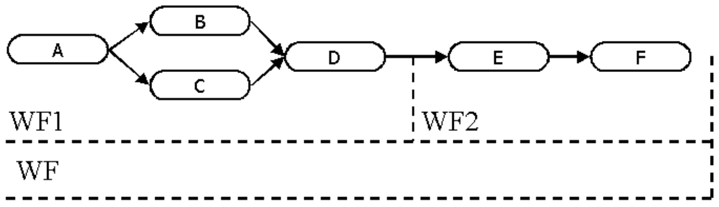


Fig. 5.25. Additivity property for workflows

A very simple example for this case is shown in Fig. 5.24. We call this property **the monotony** of the objective function.

Additivity. This property express the fact that the value of a function is the sum of the contributions of each term that is part of the function argument: $F(A+B) = F(A)+F(B)$. Again, we cannot define this property for workflow scheduling, since we do not have an addition operation on workflows set. Instead, we intuitively observe that if two distinct workflows graph WF_1 and WF_2 are glued together (sequentially ordered) to form a bigger workflow then $F(S_{opt}(WF, M)) = F(S_{opt}(WF_1, M)) + F(S_{opt}(WF_2, M))$ (Fig. 5.25). The necessary condition for the equality is that all WF_1 activities finish (are synchronized at the end of WF_1) before the activities in WF_2 can start (Fig. 5.26). We call this property **the additivity** of F_{sched}.

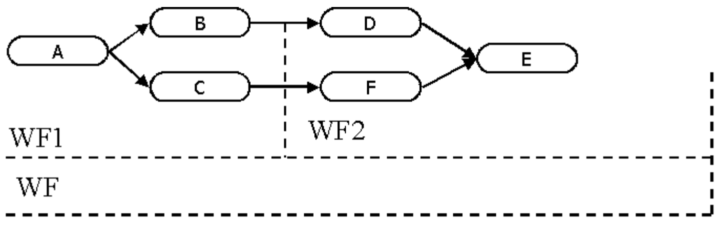


Fig. 5.26. Additivity. The contributions of the workflows cannot be clearly separated.

The two properties of F_{sched} function are essential for our dynamic scheduling strategy described in Section 5.4.2. Remember that for each workflow with loops and branches WF, we define using successive transformations, the series:

$$WF \rightarrow WF_{m_0} \rightarrow WF_{m_1} \rightarrow ... \rightarrow WF_{m_n}$$

and the associated series of DAGs:

$$DAG(WF_{m_0}) \rightarrow DAG(WF_{m_1}) \rightarrow ... \rightarrow DAG(WF_{m_n})$$

Note that each $DAG(WF_{m_i})$ is a subpart of the next element in the series - $DAG(WF_{m_{i+1}})$, and the second one is obtained from the previous by adding a set of new settled activities. The series of DAGs could be seen as an ordered set of DAGs, and the monotony property can be considered. The monotony property indicates that the length of schedule is increased at each transformation step: $F^*_{sched}(DAG(WF_{m_i}), M) \leq F^*_{sched}(DAG(WF_{m_{i+1}}), M)$. The additivity property indicates that the increase is due to the new added settled activities. However, there is no clear scheduling delimitation between the old and the new activities, which means that the new added settled activities in $DAG(WF_{m_{i+1}})$ still depend on the termination of activities in $DAG(WF_{m_i})$. Therefore, the contribution to the makespan of the activities in $DAG(WF_{m_i})$ and that of the new added activities in $DAG(WF_{m_{i+1}})$ cannot be clearly separated (Fig. 5.26).

Our intention is to generalize the scheduling strategy presented in Section 5.4.2 for any objective function that has the monotony and additivity properties. We will analyze several objective functions with these properties in the following sections.

5.7.2 Economical Cost Model

Economical models for resource allocation in Grid computing environments represent a promising research area. For such a model, the resources are associated with costs of utilization, while the users have budgets to spend for running distributed applications. We propose a simple economical model for resource allocation and a scheduling strategy similar to that described in Section 5.4.2.

Resources. Each resource in the machine set $M = \{1, 2...n_M\}$ is associated with a cost. The **resource utilization cost** $cost(m)$ can be expressed in units such as processing cost-per-FLOP, cost-per-job, CPU cost-per-time unit or any other similar metrics. Economical models of higher complexity may include other costs (e.g. for disk usage or memory usage). We assume that the applications are charged per time unit for the utilization of the resource, and the cost represents the charge for one time unit (e.g. second) of execution onto the associated resource. At the same time, a benchmark may be used to evaluate the performance of computing resources, as a *computational* *power factor* denoted by $pwf(m)$. This resource parameter can be expressed in FLOPS-per-time unit.

Communication. We assume that the links between the resources are associated with **communication cost** or network utilization cost. We denote the cost of sending 1 MB (one data unit) of data from m_1 to m_2 by $ccost(m_1, m_2)$. A simplified economical model may assume a unique communication cost for any pair of machines (m_1, m_2). Even more, we may consider an economical model, for which the communication of data is not associated with costs.

Activities of the workflow. The set of workflow tasks (activities) is $Act = \{1, 2...n_{Act}\}$, and we assume that the ECT matrix is provided. Therefore, we can calculate the cost of executing an activity t onto a resource m as $cost_{ex}(t/m) = exec(t/m) * cost(m)$.

Data links. $Data = (data(t_1, t_2))$ is a matrix $n_{Act} \times n_{Act}$ for data transfer size (in data units- e.g. MB) between the activities. A non-zero element of the matrix is associated with a workflow data-link. Therefore, we can calculate the cost of transferring data between two workflow activities t_1 and t_2 mapped onto the resources m_1, respectively m_2 as $cost_{comm}(t_1/m_1, t_2/m_2) = data(t_1/t_2) * ccost(m_1, m_2)$.

Objective function. The objective function (to be minimized) is chosen to be the total cost of executing the workflow $F(sched) = Cost(sched)$ calculated as:

$$Cost(sched) = \sum_{t \in Act} cost_{ex}(t/m_t) + \sum_{(t_1,t_2) \in DEdges} cost_{comm.}(t_1/m_{t_1}, t_2/m_{t_2})$$

where $sched(t) = (m_t, start_t)$.

Again, we denote with $S_{opt}(WF, M)$ the function which associates to a workflow WF and a set of resources M, its optimal schedule. We notice that we have the **monotony** and **additivity** properties for $F(S_{opt})$ function like in the case of *makespan* objective function:

- Monotony. It is obvious that if a workflow is extended with new activities and data links, then its optimal resource utilization cost grows.
- Additivity. If two workflows are sequentially ordered to form a larger workflow, the cost of their sum is the sum of their costs.

Therefore, we may apply the dynamic scheduling strategy described in Section 5.4.2 for scheduling workflows with loops and conditional branches.

We define a series of DAGs, using successive transformations:

$$DAG(WF_{m_0}) \rightarrow DAG(WF_{m_1}) \rightarrow ... \rightarrow DAG(WF_{m_n})$$

The workflow-scheduling problem is transformed into a set of DAG-scheduling problems, for which we may apply a static scheduling algorithm, such as a list scheduling heuristic. For example, the modified HEFT algorithm presented in Section 5.6 may be easily adapted for the new objective function, by replacing the time values with resource cost values, and by computing the priorities using exclusively these values. However, in order to compute the start times for each activity, the time values are used, but minimizing the execution length is no longer an objective of the scheduler.

5.7.3 Alternative Objective Functions

Some other, less utilized, objective functions are considered by Prodan et al. [75]. We shortly describe them in this section. However, one thing they have in common is that the monotony and additivity properties no longer hold in their cases. This means that the strategy in Section 5.4.2 cannot be applied for optimizing them. Moreover, we did not find interest for DAG-scheduling that optimizes these objective functions in related work, and therefore we do not investigate them further.

Speedup

We define the speedup of a schedule *sched* as:

$$Speedup(sched) = \frac{Makespan_{seq}(sched)}{Makespan(sched)}$$

$Makespan_{seq}$ is defined as the length of the schedule that maps the workflow WF onto a single resources. Choosing a slower resource, we obtain larger speedup values, and therefore the value has to be multiplied with a factor which gives an average computational factor $\sum_{m \in M} cpow(m)/|M|$:

$$Makespan_{seq}(sched) = Makespan(sched_{m_0}) * \frac{\sum_{m \in M} cpow(m)}{|M| * powf(m_o)}$$

The $powf(m)$ values represent the **computational power factor** associated with the resource.

The goal of the scheduling should be to maximize the speedup.

Efficiency

We calculate efficiency by dividing the speedup by the number of resources used:

$$Eff(sched) = \frac{Speedup(sched)}{|M|}$$

The goal of the scheduling should be to maximize the efficiency. Speedup and efficiency are the most common ways to report the performance of a parallel algorithm.

Synchronization Cost

Synchronization cost is measured in correlation with the tasks on the critical path (according to the definition in Section 5.4.1). We compute this value as:

$$Sync(sched) = Makespan(sched) - length_{sched}(CP)$$

where length of the CP is

$$length_{sched}(CP) = \sum_{T \in CP} exec(T/m_T) + \sum_{T_1, T_2 \in CP} comm(T_1/m_{T_1}, T_2/m_{T_2})$$

Some scheduling algorithm may be interested to minimize this value.

Load Balance

First, we define the load on a node for a schedule *sched* as:

$$Load_{sched}(m) = \sum_{T \in Act, sched(T)=(m, s_T)} exec(T/m)$$

The *Load Balance* is due to the uneven work and is calculated as:

$$LB(sched) = \frac{\sum_{m \in M} Load_{sched}(m)}{|M| * \max_{m \in M}(Load_{sched}(m))}$$

The goal should be to maximize the *Load Balance* value.

Total Overhead

Overhead is defined by Amdahl's law [76] and calculated as:

$$\mathcal{O}(sched) = Makespan(sched) - Makespan_{seq}(sched)/|M|$$

The goal should be to minimize the $\mathcal{O}(sched)$ value.

Loss of Parallelism

The *Loss of parallelism* measures the overhead, which is not due to the synchronisation:

$LP(sched) = \mathcal{O}(sched) - Sync(sched)$

Some scheduler may be interested to minimize this value.

Note that the objective functions described in this section are generally used for measuring the performance of the parallel applications. The workflow applications are just a particular case of parallel applications, and therefore their performance may be evaluated with these metrics as well. At the same time, these objective functions do not have the **monotony** and **additivity** properties, as defined in the Section 5.7.1. Consequently, the incremental scheduling strategy presented in Section 5.4.2 is useless in these cases.

5.7.4 Scheduling with Multiple Objective Functions

Let's analyze the following scenario. Suppose that resources are associated with utilization costs, as in the previous section. A user has a workflow application and a budget to run his application, from which he wants to spend as little as possible. At the same time, he is interested in running his application in a relatively short time. It is clear that the two objectives may conflict with each other, since faster resources, which can execute workflow activities in shorter times, are commonly more expensive.

Two scenarios are equally possible: The user may want to execute the application as fast as possible, but within a budget limit, or he may want to reduce the cost as much as possible, but within a time limit (deadline) [77].

```
 1. $cnt := 1$
 2. $D := DAG(WF_{m_1})$ the first partial DAG in the series of DAGs, build as
    described in Section 5.4.2
 3. Repeat until there are no more activities to schedule
 4. begin
 5.     Compute static schedule $sched_D$ for $D$ to minimize execution time.
 6.     Compute $cost(sched_D)$.
 7.     If $cost(sched_D) > budget$.
            send acknowledgement and suspend
 8.     If $cost(sched_D) * (1 - r_f) * n_{ev} > budget$.
            send warning and suspend
 9.     Schedule and execute until next scheduling event.
10.     Update $D := DAG(WF_{m_{cnt+1}})$, $cnt := cnt + 1$
11.     If $cnt > n_{ev}$ update $n_{ev} := n_{ev} + 1$
12. end
```

Fig. 5.27. Scheduling with budget limit

Since the two scenarios are similar, we analyze only the first case. First, we could observe that no matter how good the scheduling strategy and how large the budget are, it is not sure that the application execution cost will be within the limit. This can be easily demonstrated by a simple workflow application with one activity within a while-loop. Assuming that the number of iteration cannot be determined at compile time and it can be quite large, it is clear that the cost can go above any acceptable value. Therefore, the user needs to know as early as possible that the cost of his application will exceed the budget, in order to take a decision: either to raise the cost limit, or to cancel the rest of the application. We propose a scheduling strategy that deals with this scenario.

We assume that, as an additional parameter for the scheduling, we can obtain n_{ev} - an estimation of the number of the scheduling events, which cause the update of $DAG(WF_m)$ (e.g. reaching a conditional branch or termination of a loop iteration). In addition, we assume that the user is willing to take a certain risk, quantified by a **risk factor** r_f with values in [0,1]. 0 means that no risk is taken and the execution is suspended as soon as it is predicted that the budget will be exceeded. 1 means that the execution continues anyway, as long as the cost is within the limits.

The algorithm is shown in Fig. 5.27. Note that the algorithm may determine that the budget is already exceeded by the current schedule, in which case it sends an acknowledgement and suspends in step 7, or it can estimate that the schedule will probably exceed the budget with a risk factor of r_f, in which case it sends a warning and suspends in step 8.

5.8 Summary

In this chapter, we have described a new framework to schedule distributed workflow applications in JavaSymphony. JavaSymphony offers a graph-based modelling user interface to compose workflows. The graphical representation of the workflow is based on the UML activity diagram enhanced with specific elements for loops. The workflow model is explained in Section 5.2.

The graphical representation of a workflow application is stored using a simple, yet expressive XML-based specification language. A build-in scheduler uses the workflow representation to run the distributed workflow applications on a set of resources, with the support of the JavaSymphony Runtime System. This functionality is explained in Section 5.3.

We have built a scheduling model for workflow distributed applications and proposed a technique to manage the conditional branches and the loops in Section 5.4. Based on this technique, we have built two algorithms for scheduling workflows with loops and conditional branches:

- In Section 5.5 we have proposed a scheduling algorithm based on Min-min, which is one of the heuristics used to schedule meta-tasks onto heterogeneous resources;
- In Section 5.6 we have presented an algorithm based on HEFT, one of the most important DAG-scheduling algorithms, which uses the well-known list-scheduling technique.

Finally, in Section 5.7, we have analyzed several alternative scheduling objective-functions and we have described how our scheduling technique can be used for some of these objective-functions.

6 The JavaSymphony Resource Broker

In Section 5.3, we have introduced the resource broker as part of the generic scheduling process. We have defined the resource brokerage as the part of the scheduling/enactment process that finds the suitable computing resource for each of the workflow activities, based on the specified resource constraints.

In the previous sections, we have assumed that the scheduling algorithms use a static set of resources $M = \{m_1, m_2 ... m_{n_M}\}$, and determine a near-optimal mapping of the activities onto these resources. However, in real life, this is hardly true, since resources may crash, or become available at random times. Moreover, resource performance may largely vary in time, such as suitable resources become unsuitable and vice versa, thus affecting the scheduling performance. In this part of the thesis, we introduce and discuss a theoretical model for the resource brokerage, which deals with these aspects.

6.1 Modelling the Resources

We consider $M = \{m_1, m_2 ... m_{n_M}\}$ the set of all resources that may be used. The resources are associated with a set of attributes: $Att = \{att_1, att_2, ... att_n\}$.

Each attribute att_i associates to each resources an attribute value (numeric or string) in the values set. If the attribute is a dynamic attribute (e.g. system load, idle times, and available memory), this value varies in time and the attribute is defined as a function of the machine and the time:

$$att_i : M \times T \rightarrow Values, att_i(m, t) = v$$

If the attribute is static (e.g. machine name, operating system, peak performance parameters), the attribute is defined as a function over M only:

$$att_i : M \rightarrow Values, att_i(m) = v$$

6.2 Modelling the QoS for Workflow Activities

The workflow activities are associated with a set of constraints, as defined in the workflow specifications.

We denote the set of constraints by $\mathbf{C} = \{c_1, c_2, ...\}$. Each constraint c_i is uniquely associated with a resource attribute, which we denote by $att(c_i)$. A constraint is a Boolean function $c_i : M \times Time \rightarrow \{true, false\}$, which determines if a property of the attribute $att(c_i)$ holds or not. Practically, the Boolean value of $c_i(t)$ is determined by comparing $att(c_i)$ with a threshold value. For example, we may have a constraints $c(m, t)$, which takes the value of the predicate "$att(c)(m, t) \geq v_0$", if the associated attribute $att(c)$ takes numeric values.

For a workflow WF, each task $T \in Act$ is associated with a (finite) set of constraints denoted by $\mathbf{C}(T) = \{c_{(T,1)}, c_{(T,2)}...\}$.

Using these notations, we define what is a **suitable resource** for a workflow activity.

Definition 6.1. *We call a resource $m \in M$* **suitable** *for the activity $T \in Act$ at time $t \in Time$, if $c_{T,i}(m, t) = true$ for any $c_{T,i} \in \mathbf{C}(T)$.*

Definition 6.2. *For $m \in M$,$T \in Act$ and $t \in Time$, the predicate:*

$$S(m, T, t) = \wedge_{c_{T,i} \in \mathbf{C}(T)}(c_{T,i}(m, t))$$

is called the **suitability-predicate** *of the resource m for the activity T (at t).*

The resource broker has to find all the suitable resources, for all the activities of the workflow, at any moment in time. In other words, a resource broker provides a function:

$$\mathbf{B}(T, t) = \{m | c_{T,i}(m, t) = true, \forall c_{T,i} \in \mathbf{C}(T)\}$$

that determines at each moment t which resources are suitable for a workflow activity T.

Determining the suitability of the resources at each moment is not feasible: On the one hand, it does not make sense to measure the system parameters continuously, since this may lead to performance problems. Instead, the dynamic system parameters could be updated at regular intervals (as done by the JavaSymphony middleware), and consequently the **resource suitability predicate** would be updated at discrete times, too.

On the other hand, a relaxed scheduling policy may not require up to date information about resource suitability. We present three scheduling scenarios for using resource suitability information:

- The resource suitability is used only at the start of the scheduling, for the initial mapping. The advantage of this policy is obvious - a set of suitable resources is assigned only once to each tasks, and there is no need for complex resource monitoring. However, significant changes of the system dynamic parameters could dramatically deteriorate the performance.

- The resource suitability is continuously updated, and the scheduler is notified in case of changes. The scheduling complexity is increased, but adaptive decisions, which prevent the performance deterioration, are possible.
- We have adopted a hybrid scheduling policy for JavaSymphony workflow applications. The idea is to use two sets of constraints for each activity: The first set is used to determine the initial suitability of the resource. Optionally, a second set of constraints, which we call **performance contract**, is used during the execution of the workflow activities on resources. If the suitability-predicate associated to the activity performance contract does no longer hold, the scheduler may suspend the activity execution and migrate it to another suitable resource.

6.3 The Resource Availability

In dynamic computing systems such as computational grids, the resources may become unavailable or available randomly. For a distributed application, it is important to determine when a resource crashes and to recover and continue the execution after such an incident. One of the functions of the resource broker is to monitor the resources in order to determine if they are still available or not. The availability of a resource can be expressed by a function of the resource and the time:

$$Avail(m,t) = \begin{cases} true, \text{ if m is alive at t} \\ false, \text{ if not} \end{cases}$$

In combination with the suitability-predicate, we obtain a function, which tells us if a resource m may be used by one activity T, at time t:

$S(m,T,t) \wedge Avail(m,t)$ **iif** m may be used by T at t.

and the function of the resource broker is modified to include the availability of the resources as well:

$$\mathbf{B}(T,t) = \{m | S(m,T,t) \wedge Avail(m,t) = true\}$$

In practice, the resource broker does not calculate such a function, but provides an ordered series of time intervals: $I(T,m) = \{I_1, I_2 ..., I_j..\}$ such as:

$I_j = [ts_j, tf_j]$ with $ts_j < tf_j$, $Avail(m,t) = true$, and $S(m,T,t) = true$ for any $t \in I_j$.

Moreover, the resource broker is not able to forecast future intervals, but updates the $I(T,M)$ sets on scheduling events (e.g. resource becomes (un)available, or resource does no longer fulfils the suitability-condition).

6.4 Scheduling with Advanced Reservation

Scheduling with advanced reservation can be also modelled by time-intervals series. We assume that there are two levels of scheduling: high level scheduling mechanism, which supports the coordination of all the activities in the workflow, and a lower level scheduling mechanism implemented by a local resource scheduler. The scheduling is the result of the negotiation between the two scheduling mechanisms [78]:

- The local lower-level scheduler manages the queues of tasks at the resource level. At the same time, it provides information about resource performance (e.g. system parameters). It is desirable to have a resource-level scheduler, which can manage the reservations for the resource.
- The global high-level scheduler collects the relevant data from the resource-level schedulers, and, based on that, computes a schedule for the entire workflow. The high-level scheduler manages the co-ordination of the activities, according to the workflow dependencies.

Managing a new reservation request:

1. Input S_m and $R = [ts, ts + \Delta t]$
2. If R_k is not **covered** by S_m **reject reservation request**.
3. Otherwise **accept request** and assign $S_m := S_m - (R)$

Removing a reservation (cancel or finish task):

1. Input S_m and $R = [ts, ts + \Delta t]$
2. Update $S_m := S_m + (R)$.

Fig. 6.1. Managing reservation requests.

If the resource-level scheduler supports advanced reservations, the reservations and the availability of the reserved resources can be expressed with time-intervals.

Let $\mathcal{I} = \{[ts, tf] | ts < tf\}$ be the set of all the time-intervals. We add a partial order relation over $\mathcal{I}$: $[ts_1, tf_1] < [ts_2, tf_2]$ **iff** $tf_1 \leq tf_2$.

Then, an ordered series of interval $S = (I_1, I_2 ... I_j ..)$, $I_1 < I_2 < .. < I_j < ...$ is used to represent the availability of a resource m.

We denote by $\mathcal{S}$ the set of time-interval order series.

Definition 6.3. *Several operations and a partial order can be defined over $\mathcal{S}$:*

- **Addition** *of two ordered series of intervals is defined as:* $S_1 + S_2 = (I_1, I_2, ...)$ *with* I_j *either in* S_1*, in* S_2 *or union of two intervals from* S_1*, respectively* S_2*.*

- **Intersection** *is defined as:* $S_1 * S_2 = (I_1, I_2, ...)$ *with* I_j *intersection of two time-intervals from* S_1*, respectively* S_2*.*
- **Difference** *is:* $S_1 - S_2 = (I_1, I_2, ...)$ *with* I_j *time interval obtained by cutting the intervals in* S_2 *from one interval in* S_1*.*
- *We say that* I_0 *is* **covered by** S_1 *if there is* $I_k \in s_1$ *such as* $I_0 \subseteq I_k$*. We write this as:* $(I_0) < S = (I_1, I_2, ...)$*. Note that if we subtract* $I_0 = [ts_0, tf_0]$ *from* S_1*, and* $I_0 \subseteq I_k \in S_1$*, then* $I_k = [ts_k, tf_k]$ *is transformed into maximum 2 intervals:* $I'_k = [ts_k, ts_0]$ *and* $I''_k = [tf_0, ts_k]$*, whilst the rest remains unchanged in* $S_1 - (I_0)$*.*
- **Partial order.** *For* S_1 *and* S_2 *we have* $S_1 < S_2$ **iif** *each* $I \in S_1$ *is covered by* S_2*.* $<$ *is a partial order relation over* $\mathcal{S}$ *.*

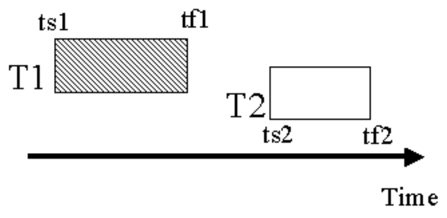
The starting time ts and the execution time $\Delta t = exec(T/m)$ are used to represent a reservation request as a time interval $R = [ts, ts + \Delta t]$. A reservation-based local scheduler manages a list of time-intervals $S_m = (I_1, I_2, ...)$, for which the resource is free and may receive reservations.

The reservations are managed by a simple algorithm (see Figure 6.1). The higher-level resource broker should be able to query the low-level schedulers for their time-interval lists S_m. For one resource $m \in M$, S_m is combined with the availability series $I(T, m)$ in a intersection of interval series: $S_m * I(T, m)$, which indicates the time frame(s) when the resource may be used by the activity T. The higher-level scheduler uses this information to build a schedule for the entire workflow.

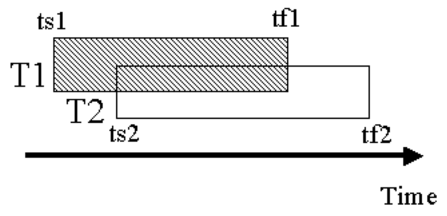
6.5 Resource Monitoring

Good scheduling decisions depend on accurate estimations for execution times and communication times for a distributed application. Therefore, it is necessary to monitor resources for performance changes, in order to update these estimations. The resource broker is responsible to collect the performance data from the resources and to update the estimations for the execution/communication times.

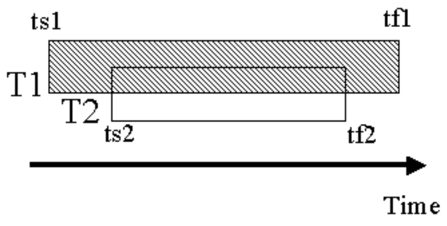
In grid environments, computing resources may be **shared**, and two distinct applications may obtain the same resource concurrently. In this way, one application may slow-down or it may degrade the performance of the other, thus diminishing the performance of both. JavaSymphony uses the share of the computing resources. Distributed objects, possibly from distinct distributed application, may reside on the same machine. The **shared access** model contrasts with the **exclusive access** to the resources, in which a single tasks at a time uses a resource, while other tasks wait in a queue. Even if the first one is more flexible, the second access model allows more accurate performance estimations. The schedulers previously presented implicitly assume an exclusive access to the resources for the activities of a workflow and static estimations of the execution times.



(a) No sharing



(b) Sharing (1)



(c) Sharing (2)

Fig. 6.2. Two tasks sharing a resource

In the following, we analyse how the execution times are affected by sharing a resource (Figure 6.2) and propose an algorithm for updating these estimations.

We assume that the sharing is **fair** (i.e the CPU cycles are equally distributed among the task sharing the same resource). For two activities T_1 and T_2 mapped onto a resource m, execution times in exclusive mode are estimated as $exec(T_1/m)$ and respectively $exec(T_2/m)$.

Assuming that T_1 starts earlier than T_2, at ts_1, if it exclusively uses the computing resource (Figure 6.2(a)), then it is supposed to finish execution at $tf_1 = ts_1 + \Delta t$, with $\Delta t = exec(T_1/m)$. As we will see, Δt varies in case of shared access to the resource.

In Figure 6.2(b), T_1 starts and finishes earlier than T_2. We assume that in $[ts_2, tf_1]$ both tasks are slowed down by 50%. Therefore, we compute the execution times for shared access as:

$\Delta t_1 = tf_1 - ts_1 = (ts_2 - ts_1) + 2 * (exec(T_1/m) - (ts_2 - ts_1))$

which is:$\Delta t_1 = exec(T_1/m) + (exec(T_1/m) - (ts_2 - ts_1))$. The second part of the sum represents the overhead due to shared access.

For task T_2, we have:
$\Delta t_2 = tf_2 - ts_2 = 2 * (exec(T_2/m) - (tf_2 - tf_1)) + (tf_2 - tf_1)$
which is:$\Delta t_2 = exec(T_2/m) + (exec(T_2/m) - (tf2_2 - tf_1))$

```
Managing a new task $T_{n+1}$, that starts at $ts_{n+1}$:
    1. Add $T_{n+1}$ to $\mathcal{T}$, $rt_{n+1} := exec(T_{n+1}/m)/(n+1)$.
    2. For each j=1,2..n update $rt_j := rt_j * n/(n+1)$
    3. Update $t_{upd} := t_{crt}$
Removing a task $T_n$ (cancel or finish):
    1. Remove $T_n$ from $\mathcal{T}$
    2. For each j=1,2..(n-1) update $rt_j := rt_j * n/(n-1)$
    3. Update $t_{upd} := t_{crt}$
Remaining execution time for $T_i$ is $rt_i + t_{crt} - t_{upd}$, which is passed to the
resource broker.
```

Fig. 6.3. Estimating execution times for shared-access.

In the case of Figure 6.2(b), T_1 starts earlier and finishes later than T_2. We assume that in $[ts_2, tf_2]$ both tasks are slowed down by 50%. Therefore, we compute the execution times for shared access as:
$\Delta t_2 = tf_2 - ts_2 = 2 * exec(T_2/m)$
and for T_1:

$$\begin{aligned} tf_1 - ts_1 &= (ts_2 - ts_1) + 2 * (exec(T_1/m) - (ts_2 - ts_1 + tf_1 - tf_2)) + (tf_1 - tf_2) \\ &= (tf_1 - ts_1) + 2 * (exec(T_1/m) - (tf_1 - ts_1 - \Delta t_2)) - \Delta t_2 \\ &= 2 * exec(T_1/m) + \Delta t_2 - \Delta t_1 \end{aligned}$$

which gives us:
$\Delta t_1 = tf_1 - ts_1 = exec(T_1/m) + \Delta t_2/2 = exec(T_1/m) + exec(T_2/m)$

As we can see, the estimation of execution times on a resource in case of shared access requires a complex analysis at resource-level.

We propose an algorithm (Fig. 6.3), which dynamically updates the estimations of the execution times for the activities that share a computing resource. The algorithm manages a list of tasks $\mathcal{T} = (T_1, T_2, ..., T_n)$, running on a single resource. Each task T_i is associated with a value rt_i, which indicates the execution time left for the task. The t_{upd} indicates the time when the last update was performed, whilst the t_{crt} is the current time.

6.6 Summary

A resource broker is essential for scheduling distributed applications in heterogeneous environment. A resource broker determines which resources are available and suitable for a workflow activity, and may support more advanced features like reservations and dynamic updates of the estimated task execution times.

In this chapter, we have built a theoretical model to describe the functionality of the resource broker. The basic elements of the model are the resources, which are associated with attributes, and (workflow) activities, which are associated with constraints. The resource broker determines the **suitability** and the **availability** of the resource, which is modelled by associated predicates. These can be calculated from the information provided by the JavaSymphony Runtime System, or from the workflow specification.

Moreover, we have investigated the implications of using advanced reservations and we have proposed an algorithm to dynamically update the task execution time estimations in case of shared access to the resources.

7

Experiments

In this chapter, we will demonstrate the usefulness of JavaSymphony, through a variety of experiments.

We have implemented and tested several distributed application by using the JavaSymphony programming paradigm and/or the JavaSymphony workflow scheduling model and JavaSymphony graph-based workflow composition tool. The list of experimental applications includes:

1. An application for image processing based on Discrete Cosine Transformation Algorithm (DCTA) (Section 7.1), which proves that JavaSymphony applications can achieve scalability on heterogeneous NOWs, respectively on homogeneous clusters;
2. Three variants of Jacobi relaxation algorithm (Section 7.2), which demonstrate how to implement message passing programs in JavaSymphony. Furthermore, these experiments illustrate the use of single- vs. multi-threaded objects, distributed events and barrier synchronization in JavaSymphony;
3. A Branch&Bound technique to solve a discrete optimization problem (Section 7.3). This experiment shows that JavaSymphony can address Branch&Bound problems and discuss scalability issues for this category of applications;
4. A distributed backtracking method to compute the number of distinct strings when applying a flattening operator on a planar word (Section 7.4). This experiment demonstrates that JavaSymphony can achieve very good speedup for computation-intensive distributed application;
5. A DES distributed encryption/decryption algorithm (Section 7.5). This application shows the advantages of using JavaSymphony in heterogeneous computing over two other similar programming paradigms;
6. A decision support system for portfolio and asset liability management (Section 7.7) proves that JavaSymphony is suitable for realistic optimization problems;

7. A distributed application for the well-known N-body problem (Section 7.6) demonstrates scalable behaviour of JavaSymphony;
8. Three workflow applications including Wien2K program package for performing structure calculations of solids (Section 7.9.1), modelling software to calibrate hydrological model parameters (Section 7.9.2), and a software system for generating astronomical image mosaics (Section 7.9.3). These applications illustrate the use of JavaSymphony workflow management system and the advantages of using our scheduling framework.

7.1 JavaSymphony DCTA

The Discrete Cosine Transformation Algorithm (DCTA) [79] can be used to eliminate non-essential information from images and compress digital data. This algorithm is commonly used to compress JPEG images. In order to do that, an input image is divided into square blocks of identical sizes. DCTA is then applied to individual blocks of the same size and with non-essential image information is eliminated. A reverse transformation produces a restored image without non-essential data, which resembles the original image.

We use the master/worker paradigm to encode a JavaSymphony DCTA. The master is encoded as a JavaSymphony Application that divides the image into square blocks of identical sizes (16x16, and 32x32). These blocks are grouped into jobs that are distributed to a number of workers, encoded as JS objects, which run on a set of computing resources (workstations or SMP nodes). After a worker has finished its job, it sends back the results to the master and requests a new job. The master application enables the JS object workers to execute the jobs, by using the JS asynchronous remote method invocations. The results are transferred back to the master, which then restores the image.

For the sake of demonstration, Figure 7.1 shows the most important excerpt of our JS application, omitting parts of the initialization, image processing, and exception handling. Figure 7.1 shows us only the processing part of an applet designed to draw the image before and after processing. Class *Job* is used as a container for a predefined number of blocks, JS objects that encapsulate *TransformBIG* instances reside on all workers and process the jobs assigned by the master. In order to be able to analyze the performance more accurately, we have artificially increased the processing time for one single block by 10, by repeating the associated computation ten times.

7.1.1 DCTA onto a Heterogeneous Network of Workstations

First, we describe the experiment that has been conducted on a non-dedicated heterogeneous NOW with up to 13 Sun workstations comprising 6x Sun Ultras 10/440, 1x Sun Ultra 10/333, 2x Sun Ultra 10/300, and 4x Ultra 1/140-170. All Sun Ultra workstations are connected based on 100 Mbits/sec bandwidth

```
public void startApplication(...)
{
  JSRegistry js = new JSRegistry();// ***** registration *****/
    VA c=new VA(2); VA[] n=new VA[nrnoduri]; /***** creating VA's *****/
  for(nr=0;nr < nrnoduri;nr++)
  {
    n[nr]= new VA(1);
    c.addVA(n[nr]);
  }
  obj=new JSObject[nrnoduri*nrproc];
  JSCodebase cb=new JSCodebase();
  cb.add("./JS/TransformBIG.class"); cb.load(c); // ***** loading codebase *****/
  for(i=0;i < nrnoduri * nrproc;i++)
    obj[i]=new JSObject("TransformBIG", new Object[] {new Integer(i)} , n[i%nrnoduri]);

  startProcess(nrnoduri,nrproc); //***** processing *****/
  js.unregister(); //***** unregistering *****/
}

public void startProcess(int nrnoduri, int nrproc)
{
  ResultHandle h[] = new ResultHandle[nrnoduri*nrproc];
  boolean isProcessed[] = new boolean[nrnoduri * nrproc];
  for(i=0;I < nrnoduri * nrproc;i++)
    isProcessed[i] = false;
  /***** PROCESSING ******/
  int jNod; Job myjob = new Job(jobSize); boolean no_more_jobs, all_ready;
  /***** initial load *****/
  for(jNod=0; (jNod < (nrnoduri*nrproc)) && (task < nr_tasks); jNod++)
  {
    task = makeJob(myjob, task, jobSize, nr_tasks); // **** build the job
    h[jNod] = obj[jNod].ainvoke("cosTransfJob",
      new Object[] { myjob , new Integer(blockDim), new Integer(alpha),new Integer(jNod) } );

  /***** load balancing *****/
  no_more_jobs = (task >= nr_tasks); all_ready = false;
  while(!no_more_jobs || !all_ready)
  {
    all_ready = true;
    for(jNod = 0; (jNod < (nrnoduri*nrproc) ); jNod++)
    {
      if( h[jNod].isReady() && !isProcessed[jNod] )
      {
        makeFromJob((Job)h[jNod].getResult()); //***** save results *****/
        if(!no_more_jobs)
        {
          task = makeJob(myjob, task, jobSize, nr_tasks); //***** build a new job *****/
          h[jNod] = obj[jNod].ainvoke("cosTransfJob",
            new Object[] { myjob , new Integer(blockDim), new Integer(alpha),new Integer(jNod)} );
          no_more_jobs = (task >= nr_tasks);
        }
        else
          isProcessed[jNod] = true;
      }
      if(!isProcessed[jNod])
        all_ready = false;
    }
  }
}
```

Fig. 7.1. Code skeleton of master/worker JavaSymphony DCTA

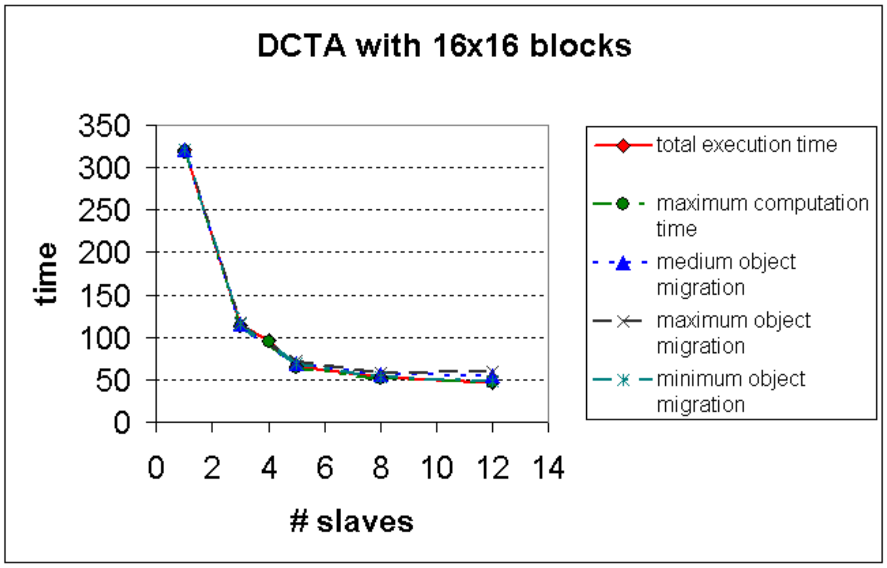


Fig. 7.2. JavaSymphony DCTA (16x16 blocks) performance on a NOW

and run Sun Solaris 8. These workstations are used by the personal for the regular work. We have used Sun's JDK 1.2.1 with a JIT compiler and native threads as the platform JVM. The job size influences the performance in two ways:

- Larger job sizes induce lower communication costs, but at the same time deteriorate load balancing;
- Smaller job sizes imply a better load balancing at the cost of an increased communication overhead.

Figures 7.2 and 7.3 show the total execution time for the entire DCTA measured at the master, and the maximum computation time of all jobs across all workers for various number of workstations and various problem sizes.

The job processing corresponds to the core DCTA without any synchronization and communication overhead. One workstation is exclusively used for the master, whereas all other workstations are processing worker jobs. We can clearly see that the maximum computation time across all workers is very close to the total execution time. Therefore, the JavaSymphony middleware produces very small overhead (e.g., communication and synchronization) for DCTA on the given workstation cluster.

We also observe that the DCTA performance scales for up to 5 workers. Thereafter, we use substantially slower workstations. The fastest workstations are up to 4 times faster than the slowest workstations. At the very end of the execution, when the master stops producing new jobs for the workers, the

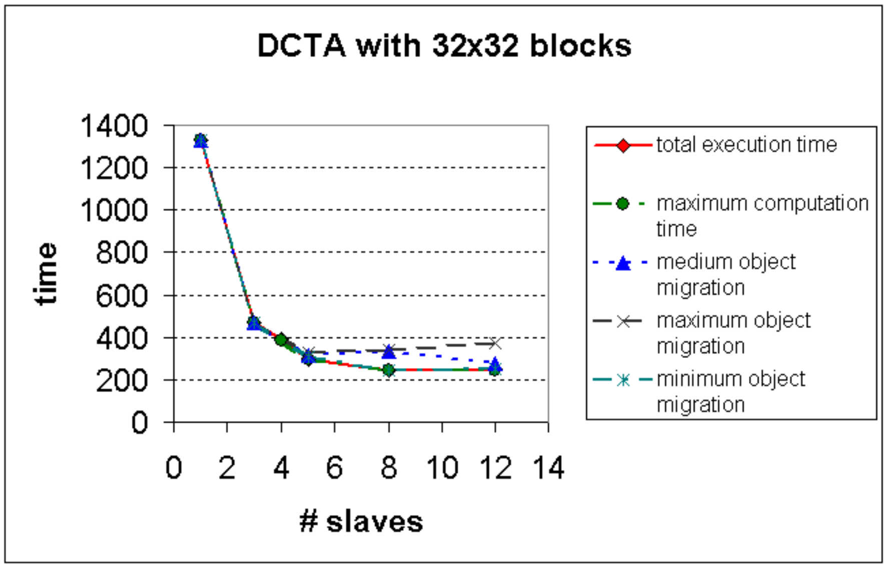


Fig. 7.3. JavaSymphony DCTA (32x32 blocks) performance on a NOW.

slowest workstations are still busy with the assigned jobs, whereas the faster workstations have already finished the work. This effect impacts the load balancing and consequently prevents also any further scaling of the DCTA for more than 5 workers on the given NOW.

Moreover, we evaluate the overhead produced by migrating objects. For this purpose we have tested 3 DCTA scenarios with different degrees of artificial migration of worker objects without changing the overall load balance: minimum migration (10-20% of the objects migrate), medium migration (50% of the objects migrate) and maximum migration (all objects migrate). Note that all migration experiments did not modify the load balance of each workstation in terms of objects per computing node. For every object migrated to a remote location, another one has been moved in its place. This policy allowed us to observe the unaltered overhead induced by object migration only, without side-effects caused by load imbalance.

The total execution times for these three scenarios are displayed in Figures 7.2 and 7.3. Due to the fact that communication costs on the given network are small compared with the computation costs, we notice that the scenarios with migration are similar to the ones without migration. This holds in particular for the problem size with 16x16 blocks. However, for 32x32 blocks, the number of jobs is rather small and the average job execution time onto different workstations varies significantly. This, in turn, aggravates the load balance in particular for the medium and maximum migration scenario.

7.1.2 DCTA onto a Cluster of SMPs

Similar experiments with DCTA have been conducted on a SMP cluster [80]. which has 16 SMP nodes (connected by FasterEthernet) with 4 Intel Pentium III Xeon 700 MHz CPUs with 1MB full-speed L2 cache and 2Gbyte ECC RAM main memory per SMP, and runs under Linux 2.2.18-SMP.

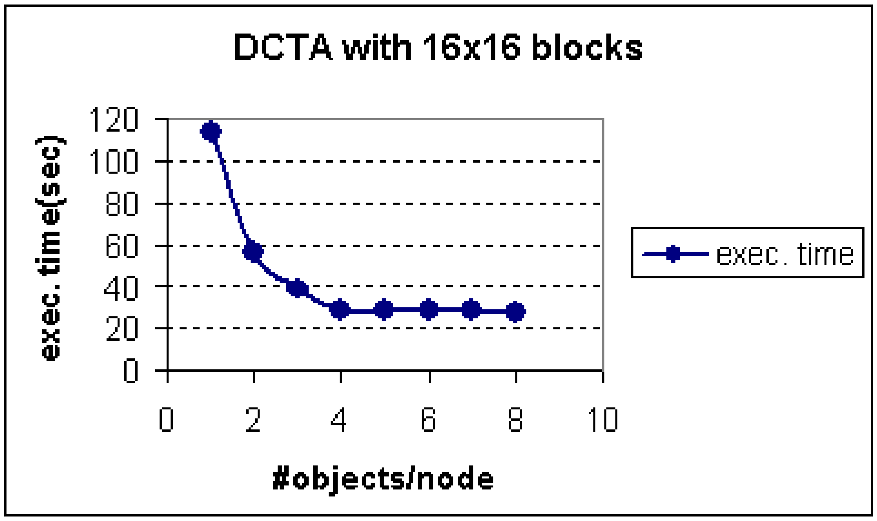


Fig. 7.4. Performance for various numbers of objects per node on a single SMP node

Each SMP node has its own IP address, and 4 CPUs share this address. The JavaSymphony Runtime System (JRS) uses the Java RMI mechanism, and implicitly the IP machine addresses to distribute objects onto the resources. Therefore it is not possible to assign JS objects to each CPU in the cluster.

Figure 7.4 shows an experiment in which we use a single SMP node to run the JavaSymphony DCTA. By increasing the number of worker objects in a single SMP node, we can substantially improve the performance. The Linux operating system presumably distributes JS objects mapped to an SMP node to its individual CPUs. Mapping more than 4 JS objects to an SMP node with 4 CPUs does not improve the performance anymore. We conclude that a number of at least 4 JS objects per each SMP node is necessary to use the computing resource most efficiently.

Figures 7.5 and 7.6 (similar to Figures 7.2 and 7.3) display the total execution time for the entire DCTA measured at the master and the maximum computation time of all jobs across all workers for various numbers of SMP nodes and distinct problem sizes.

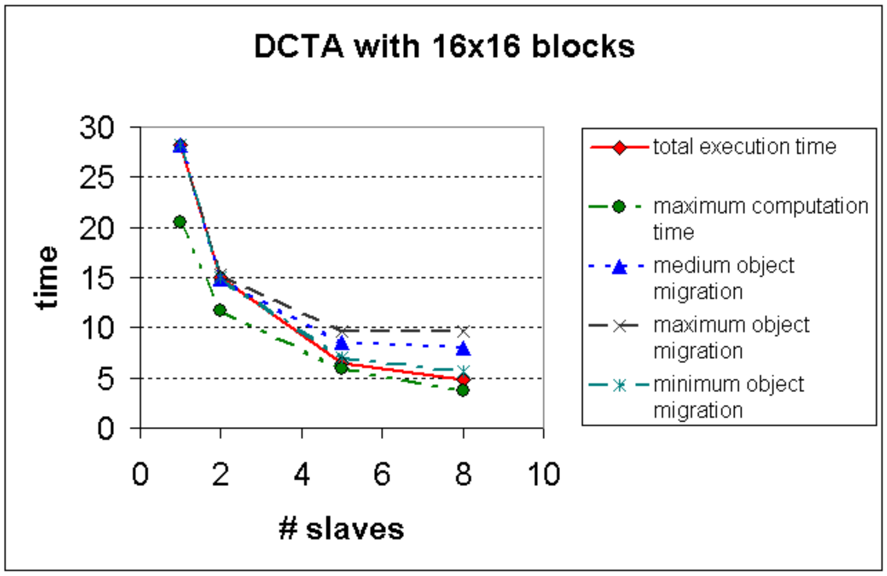


Fig. 7.5. JavaSymphony DCTA (16x16 blocks) on a SMP cluster

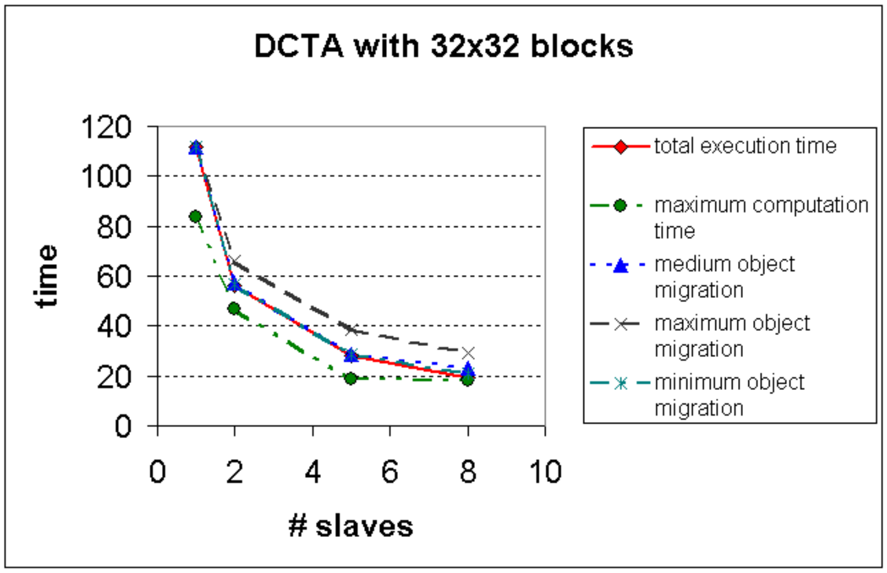


Fig. 7.6. JavaSymphony DCTA (32x32 blocks) on a SMP cluster

Note that a single SMP node corresponds to 4 CPUs. The scaling behaviour is improved compared to the experiments conducted on the workstation network, because the SMP cluster is a dedicated system that can only be used by one application at any given time. Unfortunately, the network that connects the nodes in the SMP cluster is relatively slow compared to the computing capabilities of the CPUs in this architecture. The communication overhead is significant, which produces the difference between the total execution time and the maximum job computation time across all workers. This explains why the application scales only for up to 5 nodes.

As in the previous section, we evaluate the overhead due to the object migration. For this purpose, we have tested the same 3 DCTA scenarios with the three degrees of artificial migration of worker objects, without changing the overall load balance (i.e. objects per node rate). According to Figures 7.5 and 7.6, only the minimum migration scenario produces small overhead. For medium and maximum migration scenarios, the migration impact on the performance cannot be ignored. The communication network of this architecture is relatively slow compared to the computational capabilities of the individual CPUs on the SMP nodes. Therefore, depending on the degree of migration, the overall execution time grows gradually. On the other hand, since the architecture is a dedicated homogeneous system, we achieve a close-to-perfect balance of the workload distributed among the worker JS objects.

7.2 Jacobi Relaxation

7.2.1 Jacobi Relaxation: Single- vs. Multi-threaded Objects

In order to examine whether JavaSymphony is suitable for message passing programs and to compare the performance impact of single-threaded versus multi-threaded objects, we have built a JavaSymphony version of the Jacobi relaxation [81].

The Jacobi relaxation iterative method is used to approximate the solution of a partial differential equation discretized on a grid. The algorithm consists of successive steps of computation followed by communication. We have encoded a JavaSymphony version that splits a square matrix into horizontal blocks (1-dim. row-wise distribution), which are distributed onto SMP computing nodes for processing. In order to update the matrix elements of the block assigned to an SMP node, a JS (processing) object is generated. Before computing new matrix values for the local block, non-local matrix elements stored on the upper and lower neighbouring SMP nodes are needed. Therefore, on each SMP node, two separate JS (communication) objects are created, which are responsible for communication and synchronization with the immediate neighbouring SMP nodes. Communication objects have been generated as multi-threaded JS objects.

```
// *** INITIALIZATION PART
// *** builds a communication object for upper and lower neighbour;
// *** communication objects encode synchronization
  downLocalObj = new JSNeighbour ();
  downJSObj = JSObject.convertToJSObject(downLocalObj);
  upLocalObj = new JSNeighbour ();
  upJSObj = JSObject.convertToJSObject(upLocalObj);
...
// *** ALGORITHM PART - for generic iteration
processNewIteration(){ ...
      // *** block until data for the current iter. is received from neighbours;
      // *** variable iter is used to  synchronize objects
  lowerData = (double[])downJSObj.sinvoke("getData", new Object[]
           { new Integer(iter) } );
  upperData = (double[])upJSObj.sinvoke("getData", new Object[]
           { new Integer(iter) } );
  doComputation();
  ...
// *** COMMUNICATION PART - asynchronously send data to neighbours
  downJSObj.oinvoke("send", new Object[] { new Integer(iter),  lastLine } );
  upJSObj.oinvoke("send", new Object[] { new Integer(iter),  firstLine } );
  iter++;
  processNewIteration();
... }
```

Fig. 7.7. JS Jacobi Relaxation (without JS events or JS barrier-synchronization)

The code excerpt for this implementation is presented in Fig. 7.7. A variable that indicates which iteration is currently being processed is used for synchronization. The computation phase for the new iteration is suspended until the corresponding lines from the neighbours (with the same iteration number) arrive from the lower and upper neighbours. At the end of the computation phase, the updated matrix border lines are asynchronously sent to the neighbours.

For this experiment, we compare two versions: one that uses single-threaded and another that employs multi-threaded processing objects on a SMP cluster configuration (see Section 7.1.2) with a fixed number of nodes. Overall, three (two multi-threaded communication and one processing) JS objects are placed on each SMP node. A single-threaded processing object can only use one CPU of an SMP node, whereas a multi-threaded object can use several CPUs and, therefore, can exploit intra-node parallelism. For our experiments, we use SMP nodes with 4 CPUs. The experiments are based on a fixed matrix size (1000x1000) and a fixed number of Jacobi iteration steps (100). In order to show the effect of a multi-threaded JS object, the computational load

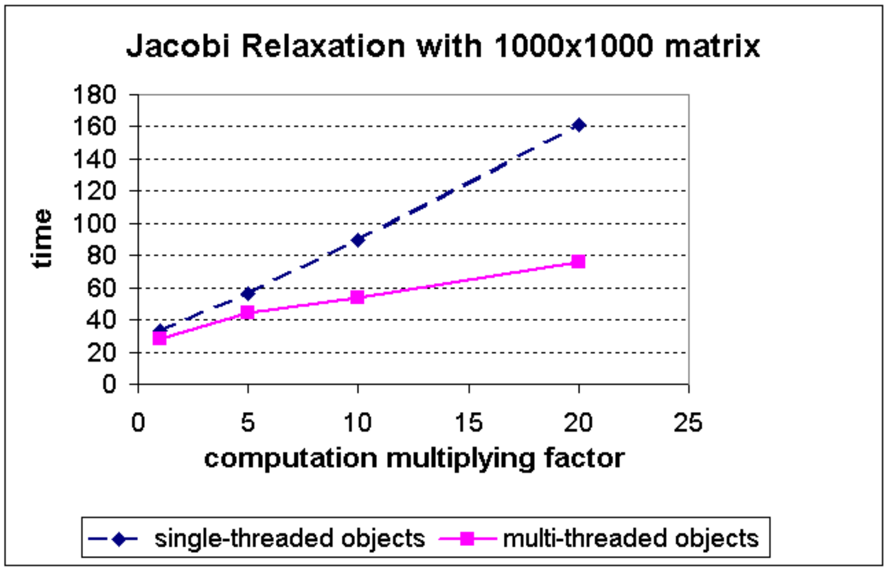


Fig. 7.8. Jacobi relaxation. Performance comparison for single- and multi-threaded JS objects

of each processing JS object is artificially modified by multiplying it with a factor ranging from 1 (corresponds to original Jacobi relaxation) to 20 (the original calculations have been repeated 20 times). Therefore, the computational load varies, whilst the communication and synchronization overhead remains constant. In this way, we can examine different computation/(communication + synchronization) ratios.

Figure 7.8 shows the total execution time for 4 SMP nodes of the JS Jacobi relaxation based on single-threaded and on multi-threaded JS processing objects, respectively. Even though the computational load is increased multiple times and we employ SMPs with 4 processors, the total execution time raises much slower, which indicates substantial synchronization and communication costs of the Jacobi relaxation implementation. In the worst case, the performance of the Jacobi relaxation can be improved by 20% by using multi-threaded JS objects. In the best case, the performance gain reaches 100 %. It is obvious that for this particular JS Jacobi relaxation implementation, JS multi-threaded objects perform visibly better than the single-threaded ones. On the other hand, by artificially increasing the computation to overcome communication/synchronization overheads, we achieve a speedup of up to 2 for multi-threading objects on 4CPU nodes. We assume that the speedup would further grow with the raise of the computation load. This experiment shows that JS is suitable for medium- to coarse-grained parallelism, but fine-grained parallelism may lead to performance deterioration.

```
// *** INITIALIZATION PART
// ***  encode producer and consumer to communicate with neighbours
   // * producer for event to be sent to the upper neighbour;
   // * event type matches with the neighbour's consumer type
 prodUp = new JSEventProducer(thisBlock,
          JSConstants.C_USER_TYPE + 2* index,
          JSConstants.C_ANY_LOCATION, null);
  // * consumer for the event produced by the upper neighbour;
  // * event type matches with neighbor's producer type
 consUp = new JSEventConsumer(thisBlock,
          JSConstants.C_USER_TYPE + 2* index -1,
          JSConstants.C_ANY_LOCATION, "processEvent" );
 consUp.register();
... // * similar for the lower neighbour
...
// *** ALGORITHM PART - generic Jacobi iteration
void processNewIteration(){ ....
   doComputation();
     // * matrix lines (lastLine, firstLine) are attached to the events as parameters
   prodUp.produceEvent( new Object[] { firstLine,
                new Integer(index), new Integer(iter) } );
   prodDown.produceEvent( new Object[] { lastLine,
                new Integer(index), new Integer(iter) } );
}
...
// *** PROCESSING EVENT
    // * parameter types match with the parameters
    // * of the method that produces events
public void processEvent(double[] line, Integer source, Integer iteration) {
    // * block until current iteration  iter
    // * matches with the iteration sent by the neighbour
  wait_until_iteration(iteration, iter);
  update_matrix_bound(source, line);
    // * test whether data from both neighbours
    // * have been updated for the current iteration;
  if( all_data_received(iter) )
  {  // start new iteration
     iter++;
     processNewIteration();
  }
}
```

Fig. 7.9. JS Jacobi Relaxation with events

Note that the programmer could mimic a multi-threaded object on a SMP node by creating several single-threaded objects. However, by doing so, programming gets more complex as more methods of different objects must be invoked to obtain the same effect as for a single multi-threaded JS object. The main purpose of multi-threaded objects is to exploit parallelism within a single object on shared memory multi-processors without the need to create multiple objects and to call methods of distinct objects.

7.2.2 Jacobi Relaxation: Events and Barrier Synchronization

The JS Jacobi Relaxation version of Section 7.2.1 requires explicit programming for synchronization. In this section, we exemplify two JS programming features, namely events and barrier synchronization, which simplify the programming effort substantially.

```
// *** INITIALIZATION PART - for each iteration a JS barrier object is built
// *** noObjects represents the number of threads that wait
for(id =0; id <maxIteration; id++)
   JSRegistry.newBarrier(noObjects, id);
...
// *** ALGORITHM PART - main iteration
void processNewIteration(){ ....
  doComputation();
    // * barrier-synchronization -- blocks until all (remote) threads reach this point
  JSRegistry.barrier(iter);
    // * communication is direct and synchronous; neighbours provide matrix lines
  objUp.sinvoke("getLastLine", new Object[] {});
  objDown.sinvoke("getFirstLine", new Object[] {});
  iter++;
  processNewIteration();
... }
```

Fig. 7.10. JS Jacobi Relaxation with JS barrier-synchronization

Figure 7.9 shows a Jacobi Relaxation code excerpt based on JS user-events for synchronization and communication. This version includes processing objects but no communication objects. JavaSymphony specific classes for event consumers/producers are used to encode synchronization and communication. The variable *iter* refers to the current iteration number, and synchronizes all objects, before proceeding with the next iteration. The variable *iter* is sent via events among processing objects. Matrix border rows are transmitted as event parameters between neighbouring objects. The *index* member of the processing object class uniquely identifies each processing object and it is used to compute the unique event-types for the neighbouring objects.

Figure 7.10 reproduces a code excerpt of a Jacobi Relaxation implementation based on JS barrier synchronization, which is the most simple version to implement. The barriers are declared in the initialisation phase. One barrier is placed at the end of each computation cycle. When encountered, each thread waits until the rest of the threads have reached this point. Thereafter, all threads continue processing by accessing the data from their neighbours via JS synchronous remote method invocation.

7.3 Branch and Bound Application

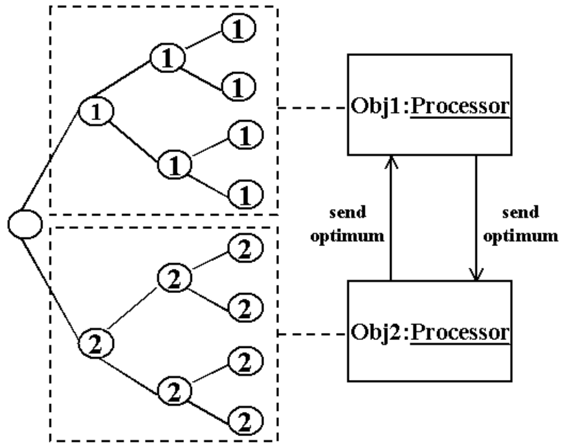
Branch&Bound [82] is a technique for solving problems by using the *divide & conquer* strategy. An optimal solution has to be found in a large space of solutions. The initial solution space is recursively divided into subspaces. As a result, an abstract searching tree is created: the internal nodes represent subspaces of possible solutions, while the leaf nodes represent the solutions. Each solution is quantified by a cost function. The solution with the optimal cost value is required. The goal of Branch&Bound technique is to exclude parts of the searching tree, which cannot provide feasible solutions with better costs than the very latest computed best-cost value.

We use Branch&Bound technique to solve a *discrete optimization problem* [83]. The goal is to search the optimal value of the cost function $f : x \in Z^n \to R$, and the solution $x = \{x_1, ...x_n\} \in Z^n$, for which the function's value is optimal. The domain of the cost function f, called the *solutions space*, is generally defined by means of a set of m constraints over the elements of the definition space. The constraints are generally expressed by a set of inequalities:

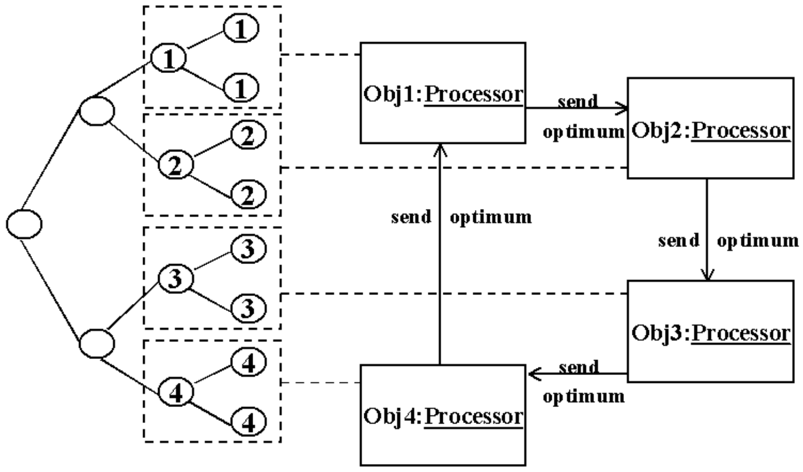
$$\sum_{i=1}^{n} a_{i,j} x_i \leq b_j, \forall j \in \{1, ..., m\}$$

For the sequential algorithm, we use the Branch&Bound strategy as described above. For the parallel implementation, the search-tree is split between JavaSymphony objects, within the nodes of a JavaSymphony level-2 VA. We denote these objects as computing objects, which work in parallel to find the solution with the optimal value. Sub-trees of the search-tree that represents the solution space are distributed to each computing object (Fig. 7.11). The set of the computing objects is organized as a ring. If a better local optimum is found by one of the computing objects, it is sent to the next object in the ring. The neighbour updates its own local optimum if necessary, and sends this value further. All the computing objects update their local best-cost values until one that has an equal-or-better value stops this transmission. This is either the one that has found the new optimal cost value in the first place, or another one that has found a better one in the meantime.

For the parallel implementation, we analyze two distinct approaches. In the first approach, equal parts of the initial search-tree are distributed to



(a) 2 computing objects



(b) 4 computing objects

Fig. 7.11. B&B solution space divided between several computing objects

the computing objects from the start, with no later redistribution, whilst the latest best-cost value is updated as explained above. In the second approach, the sub-trees are distributed among the computing objects at the beginning and, in addition, work is redistributed whenever one of the computing objects becomes idle.

A characteristic of B&B parallel algorithms is that the gain in performance depends significantly on the input data, and the execution time for various workloads cannot be predicted. The following examples explain this behaviour:

Let's assume that we split the problem between exactly two computing objects, as in Fig. 7.11(a). In the sequential version, the first sub-tree is obviously analyzed by the algorithm before the second sub-tree. For the parallel algorithm, two things could happen, if the two computing objects analyze their search sub-trees in parallel:

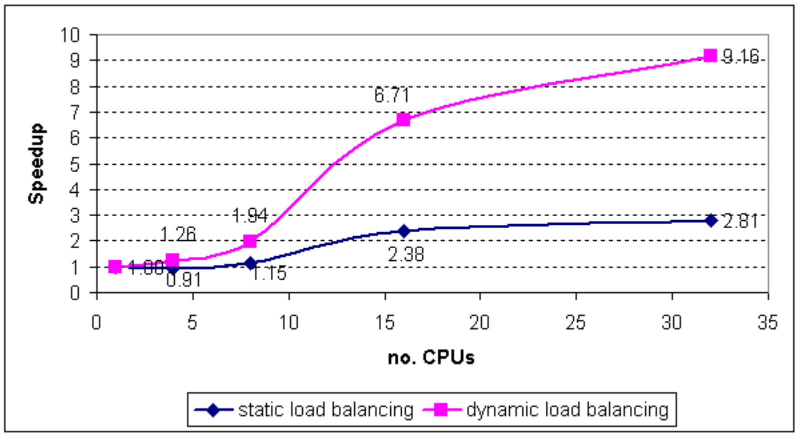
- The first computing object receives a better optimal value from the second computing object, which allows the elimination of some search paths in the first sub-tree. Consequently, the workload is smaller, and performance is improved - even super linear speedup is possible.
- In the first part of the computation, the second computing object does not have access to the best optimum value for the first sub-tree, since this has not been fully analyzed. This forces the computing object to analyze additional search paths, which would have been quickly rejected in the sequential algorithm. The workload is higher and performance deteriorates.

We have tested the JavaSymphony B&B application on a SMP cluster [80] with 16 SMP nodes (connected by FasterEthernet), each of them with 4 Intel Pentium III Xeon 700 MHz CPUs (see Section 7.1.2). We analyze the performance of the Branch&Bound algorithm in experiments for three distinct, randomly generated inputs for the discrete optimisation problems with $m = 40$ equations and $n = 10$ variables.

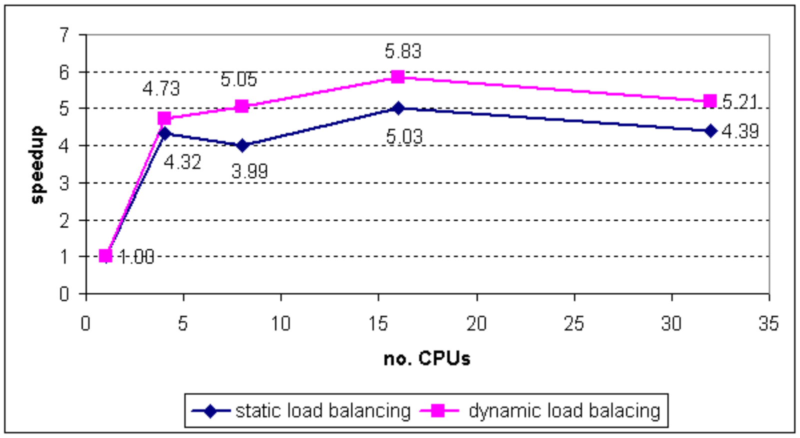
The speedup results are shown in Fig. 7.12. Performance data values for the experiments can be found in Table 7.1 and in Table 7.2. The number of visited nodes proves the variation in the computation load. Two overheads due to communication are presented: (1) the overhead produced to propagate the local optimal cost to the rest of the computing object, and (2) the overhead produced by the workload redistribution (only for the dynamic load balancing version). The difference between the total computation time and the average computation time suggests the existence of the overhead due to unequal workloads, especially for the static load balancing.

The performance issues discussed above are demonstrated by these three experiments. As we can see, performance varies, even if the problem size is the same for all three. In the first experiment (Fig. 7.12(a)), for up to 8 CPUs the speedup grows only up to 2 (in the dynamical load-balanced version). With 16 CPUs, the performance is more than 3 times better than 8 CPUs case, which represents a super linear speedup.

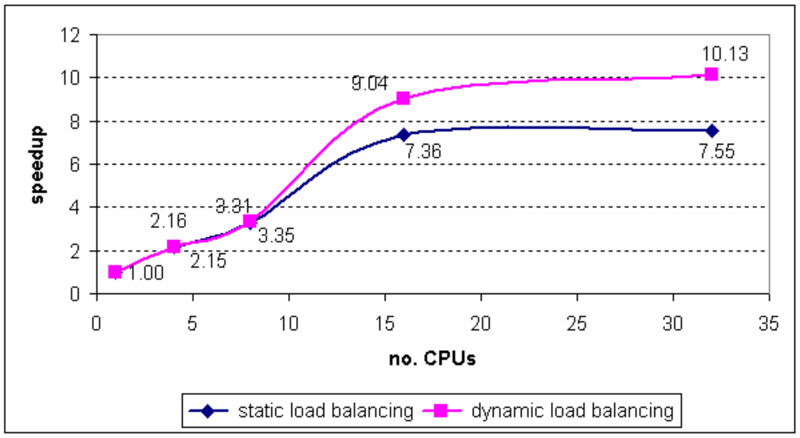
The performance variation could be explained by analyzing the values in Table 7.1: In the dynamical load-balanced approach, while the number of visited nodes grows with the number of CPUs used for up to 8 CPUs, thus deteriorating the performance, for 16 CPUs it is suddenly lower than the previous value. The bigger number of computing objects causes this behaviour, as some of the computing objects finish and find the optimal value faster. Parts of the tree are faster rejected and workload is redistributed. Observe that the number of the visited nodes is reduced for the static load balancing approach, but the large difference between the average time of computing and the total time of execution proves the unbalance of the workloads, which affects the general performance.



(a) Experiment 1.



(b) Experiment 2.



(c) Experiment 3.

Fig. 7.12. JavaSymphony Branch&Bound algorithm performance.

No. CPUs	Total time (ms)	Avg. Time (ms)	Nodes visited	Overhead 1	Overhead 2	Efficiency
PROBLEM 1						
Sequential						
1 CPU	186258.00		36665880			
Static load balancing						
4 CPUs	204817.00	152526.50	110959591	653.25	0	0.2273
8 CPUs	162590.00	105733.50	153053162	656.75	0	0.1432
16 CPUs	78108.00	34456.00	108259200	612.00	0	0.1490
32 CPUs	66272.00	23239.13	136903433	1006.16	0	0.0878
Dynamic load balancing						
4 CPUs	147980.00	147960.50	96469135	652.00	394.00	0.3147
8 CPUs	95847.00	95785.50	119274168	572.00	1334.00	0.2429
16 CPUs	27760.00	27225.69	72932293	463.69	2000.06	0.4193
32 CPUs	20340.00	19274.44	80687271	490.00	4457.44	0.2862
PROBLEM 2						
Sequential						
1 CPU	54363.00		11605272			
Static load balancing						
4 CPUs	12592.00	11412.25	8271101	1119.50	0	1.0793
8 CPUs	13634.00	12056.25	14262503	2305.88	0	0.4984
16 CPUs	10813.00	9290.69	20820747	2361.69	0	0.3142
32 CPUs	12395.00	9560.88	40836445	2387.94	0	0.1371
Dynamic load balancing						
4 CPUs	11494.00	11446.25	8324727	952.00	146.00	1.1824
8 CPUs	10759.00	10634.25	12873919	1474.75	625.38	0.6316
16 CPUs	9322.00	9081.44	18988918	1579.25	1069.25	0.3645
32 CPUs	10432.00	9426.03	36334837	1038.25	2501.06	0.1628

Table 7.1. Timing for B&B. Overhead 1 for transferring the optimum; Overhead 2 for redistributing the work

The second experiment (Fig. 7.12(b)) shows us a different behaviour: we get a significant speedup for up 4 CPUs - more then 4 times performance improvement. On the other hand, using more than 4 CPUs does not improve the performance anymore. The values in Table 7.1 could explain the phenomenon. The number of visited nodes is smaller compared with the first experiment. Even the sequential version eliminates important parts of the search tree. By using 4 computing objects, more search paths are eliminated and the speedup becomes super-linear for both versions. However, by raising further the number of computing objects and CPUs, the number of visited nodes grows as well. The overheads due to local optimum propagation (asynchronous communication) and to the workload redistribution for the dynamic load-balancing version become significant for the overall performance, since the total execution time is so small. These aspects produce performance deterioration for static load-balancing implementation at 8 CPUs and for both versions at 32 CPUs. In both cases, we can see that the number of the visited nodes gets 2 times bigger from 16 CPUs to 32 CPUs.

No. CPUs	Total time (ms)	Avg. Time (ms)	Nodes visited	Overhead 1	Overhead 2	Efficiency
Sequential						
1 CPU	352285.00		35378744			
Static load balancing						
4 CPUs	162750.00	157204.50	62515907	200.50	0	0.5411
8 CPUs	106455.00	101661.75	73955665	263.00	0	0.4137
16 CPUs	47838.00	37430.25	54204058	158.81	0	0.4603
32 CPUs	46667.00	33081.47	93405849	91.50	0	0.2359
Dynamic load balancing						
4 CPUs	163934.00	163906.75	62412733	176.50	262.00	0.5372
8 CPUs	105029.00	104863.63	73822916	190.25	1168.00	0.4193
16 CPUs	38989.00	38794.31	50727885	196.50	2488.50	0.5647
32 CPUs	34765.00	33795.06	80594720	120.75	4746.47	0.3167

Table 7.2. Performance data for B&B problem 3

In the third experiment (Fig. 7.12(c)), we obtain a steady and significant performance improvement, for up to 16 CPUs. Again, when increasing the number of CPUs from 8 up to 16 CPUs, there are actually less processed nodes in the search-tree, which produces a super linear speedup. We do not get better performance by using 32 CPUs. In the dynamic load-balancing version, the overhead for re-balancing the workload becomes significant, thus preventing further improvements.

The experiments demonstrate the unpredictability of the performance for the parallel version of the Branch&Bound algorithm. However, we can see that there is usually a performance improvement - in some cases minimal, but in some other cases significant, depending on the input data and the number of processing objects used. On the other hand, workload redistribution produces always better results than static load balancing.

7.4 JavaSymphony Backtracking Application

Backtracking [82] is a generic and very well known method of solving a large class of problems. The backtracking method is based on the systematic inquisition of the possible solutions where, through the procedure, many possible solutions are rejected before even examined. Commonly, one solution is built from partial solutions, which satisfy a set of constraints. The partial solutions are expanded gradually to a full solution. Each partial solution can provide a few choices for the next searching step. When the constraints are not fulfilled, the search path is abandoned. The goal is to find all possible solutions (or alternatively, a single solution) by searching a solution space that is scaled down by the constraints. Usually, backtracking algorithms have an exponential complexity and thus they are computation-intensive. Since at each steps a few choices for the next partial solution could exist, the parallelization of the algorithm becomes natural.

For the following experiment, we have chosen a particular problem that can be solved by using the backtracking method. However, our implementation can be generalized for a large class of computational-intensive backtracking problems. The selected problem has been inspired from the problem of computing the number of distinct strings when applying the *flattening operator* to a planar word (see [84]).

To solve this problem, we must find all distinct possibilities to move a number of n identical objects along m consecutive positions in a m-tuple (1, 2, 3 ... m). One object is moved at a time from a position i to the next position $i+1$ until all objects reside on position m, such that $1 \leq i \leq m-1$. We denote this move by $[i, i+1]$. Initially, all objects start at position 1 in the m-tuple. For example, if we have to move 2 objects along 3 positions in a tuple $(1, 2, 3)$, we can find two solutions, represented as ordered sets:

$$\{[1,2], [1,2], [2,3], [2,3]\ \} \text{ and } \{\ [1,2], [2,3], [1,2], [2,3]\}$$

The backtracking method can be applied to solve this problem in an obvious way: Each object is moved from one position to the next, until all the objects are placed onto the last position. At each step, several distinct moves are possible. Consecutive possible moves yield sub-problems that can be solved in parallel with the other sub-problems. As simple as it looks, the requested number raises very fast: for 5 objects and 6 positions we obtain 701149020 distinct possibilities to move the objects from position 1 to 6. The sequential algorithm has exponential complexity, which requires extensive computation time on any computer architecture.

We have built a JavaSymphony application for the parallel backtracking algorithm. The search is split into sub-problems, which are distributed among computing objects (JS-objects). Each computing object maintains a queue for jobs (i.e. sub-problems as defined above) that can be passed to other idle computing objects. This algorithm requires a high computational effort. Communication is low even in the case of dynamic load balancing when idle computing objects receive additional jobs from busy objects. Additional overhead occurs in the final phase of the program, when each computing object solves a sub-problem and all the queues are empty. The sub-problems have distinct computational needs, which may cause significant load imbalance and consequently deteriorate the performance.

Figure 7.13 compares the overall speedup values with the ideal speedup, for various number of CPUs, on the 4way SMP cluster (described in the

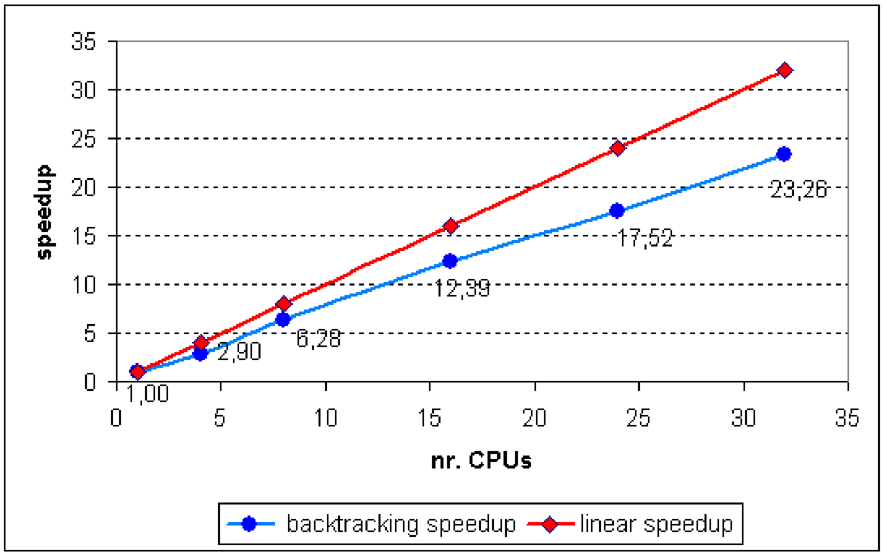


Fig. 7.13. JavaSymphony backtracking algorithm performance

previous experiments). Although we do not achieve maximal efficiency, the speedup values raise steadily with the number of CPUs. Therefore, we consider that JavaSymphony is suitable to implement parallel versions of any other computation-intensive backtracking algorithms.

No. CPUs	exec. time	exec. time (avg.)	ideal exec. time (avg.)	comp. time (avg.)	comm. time (avg.)
Sequential					
1 CPU	823785				
Parallel					
4	283777	214585	205946	214316	219
8	131249	110709	102973	110022	603
16	66474	56395	51487	55209	1032
24	47022	39599	34324	37373	1900
32	35412	31105	25743	28089	2277

Table 7.3. Various timings for backtracking algorithm. Values are represented in ms

In order to explain the performance loss, Table 7.3 displays experimental values for the total execution time, average execution time (across all computing objects), ideal execution time (theoretically computed assuming perfect speedup), computation time (spend by each object without any communication or parallelization overhead), and communication overhead (time for sending jobs to idle computing objects). All tabulated values, except total execution times, are average values across all computing objects.

Clearly, load imbalance – for reasons mentioned above - can rise noticeable for increasing number of CPUs, which is demonstrated by the difference between total execution time and average execution time. The communication overhead is reasonably small. The small difference between average computation time and average execution time shows the overhead due to the JavaSymphony middleware.

7.5 DES Encryption/Decryption

In the following, we present an experiment with the DES (Data Encryption Standard) encryption/decryption algorithm [85], to compare the performance of three Java-based programming paradigms for concurrent and distributed applications including JavaSymphony, JavaParty, and ProActive. JavaParty is centered on semi-automatic load balancing and locality control where most strategic decisions are taken by the underlying runtime system. In contrast, JavaSymphony and ProActive provide explicit user control of load balancing and locality. ProActive represents a lower-level programming paradigm that does not provide the user with an API to system information. For our experiments, we have used the publicly available JavaParty version 1.05b [86] and ProActive, version 0.9.1 [87].

The DES encryption/decryption algorithm [85] uses a key of 56 bits, which is extended with another 8 parity bits. DES tries to detect the key that has been used to encrypt a message using DES, based on a "brute-force" approach (every possible key is tested). The assumption is that we know a string that must appear in the encrypted message, which is used to test its authenticity. For the sake of demonstration, the experiments examine only a sub-set from the total number of 2^{56} possible keys.

Figure 7.14 depicts the design of the JavaSymphony DES decoding algorithm. The *DesDecoder* objects process a space of possible keys, which are provided by one or several *KeyGenerator* objects. A *DesDecoder* acquires the keys from the *KeyGenerator* through a synchronous method invocation. The *KeyGenerator* keeps track of the keys that have already been generated. After the *DesDecoders* have decoded a message by using the assigned keys, a one-sided method invocation is used to transfer the decoded message to a *TextSearcher* object, which searches it for the known string, to determine if the correct encryption key has been found. Depending on the length of the encoded message, the *DesDecoder* and *TextSearcher* components may require various computational efforts.

For these experiments, two connected Beowulf cluster architectures have been used. The first (slower) cluster has SMP nodes (connected by FastEthernet) with 2 CPUs (Pentium II, 400MHz, 512 MB ECC Ram) each. The second

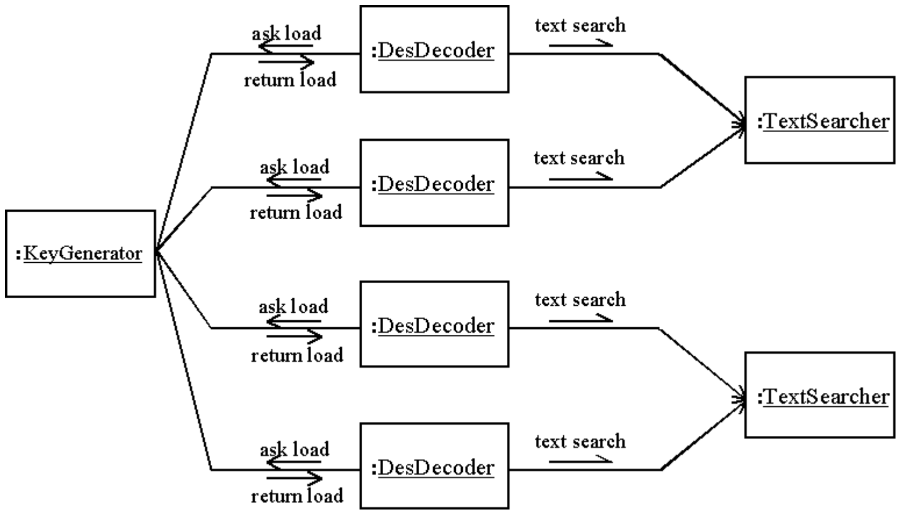


Fig. 7.14. JavaSymphony DES decoding algorithm design

(faster) cluster consists of 16 4-way SMP nodes (connected by Myrinet) with Pentium III Xeon (700MHz) CPUs and 2GB ECC RAM main memory per SMP (see Section 7.1.2). Both clusters run under Linux 2.4.17-PMC-SMP and use Sun Java 2 SDK with a JIT compiler and native threads.

We have implemented three distinct versions of the DES algorithm, one for each programming paradigm.

- **JavaSymphony version:** Two different clusters are used, one for each functional component. The computation-intensive component (*DesDecoder*) is mapped on the faster cluster, whereas the less computation-intensive component (*TextSearcher*) is mapped on the slower cluster. The *KeyGenerator* component (with very small computational overhead) is mapped together with one of the *DesDecoders*, in order to reduce communication. JavaSymphony requires the user to create a virtual architecture and to use synchronous and one-sided method invocations.
- **JavaParty version:** The second version is based on JavaParty, which introduces a new class modifier called *remote*. Only the objects that are generated based on remote classes can be distributed. In order to parallelize a program, JavaParty requires the programmer to generate specific threads, which execute methods of remote objects onto distinct nodes. On the other hand, the conversion to remote objects is quite transparent for the programmer. Therefore, we consider that the programming effort is about the same for both JavaSymphony and JavaParty. A JavaParty specific pre-compiler is used to compile JavaParty programs to Java RMI programs.

- **ProActive version:** The third version is based on ProActive. The ProActive class library offers extensive functionality, but at a rather low level. The so-called future objects and one-sided method invocations are used to parallelize DES. In contrast to JavaParty, no explicit threads have to be generated. Objects can be explicitly mapped onto a one-dimensional architecture structure (e.g. a set of connected nodes). Java objects are automatically substituted by stubs and can be locally or remotely accessed, fully transparently. However, the RMI details are not sufficiently hidden, and mapping objects requires starting RMI servers and connecting to their addresses. Therefore, we consider that ProActive programming paradigm requires more programming efforts for DES implementation.

We have managed to further improve the performance of these DES versions as follows:

- The load balancing for the *DesDecoders* can be dynamically controlled. When a *DesDecoder* finishes a job, it requests a new set of keys from the *KeyGenerator*.
- Since text-processing takes much less time compared to key processing, we make a *TextSearcher* to serve more than one *DesDecoder*.
- As *TextSearchers* are less computation-intensive, they are explicitly mapped on the slower cluster, whereas the *DesDecoders* are mapped on the faster one. JavaSymphony can determine whether a cluster is faster or slower through its high level interface to system parameters. Both JavaParty and ProActive do not support the mapping of objects based on performance information of the underlying computing infrastructure. Therefore, for the JavaParty and ProActive versions, we use a round-robin mapping strategy instead.

Figure 7.15 compares the speedup values for the three code versions for different cluster sizes. The notations *fCPU* and *sCPU*, respectively, mean that CPUs of the faster (with Pentium III CPUs) and respectively slower (with Pentium II CPUs) cluster have been used. The speedup is computed relative to a sequential version that has been run on a single CPU of the faster cluster.

JavaParty and ProActive perform only slightly better than JavaSymphony on a homogeneous cluster. This is caused by a more complex communication protocol used by JavaSymphony to hide the RMI details from the programmer. On the other hand, JavaParty uses a separate JavaParty pre-compiler to compile remote JavaParty objects into RMI remote objects, and therefore the runtime middleware overhead is reduced. ProActive reduces the middleware overhead by providing a lower-level programming paradigm.

However, if a heterogeneous cluster architecture is available, then JavaSymphony provides the capability to map parts of an application to specific subclusters. If we employ both fast and slow cluster in a single experiment, then under JavaSymphony we can place the *TextSearcher* on the slower cluster and the other components on the faster cluster. For the heterogeneous cluster architecture, JavaSymphony visibly outperforms the JavaParty and ProActive

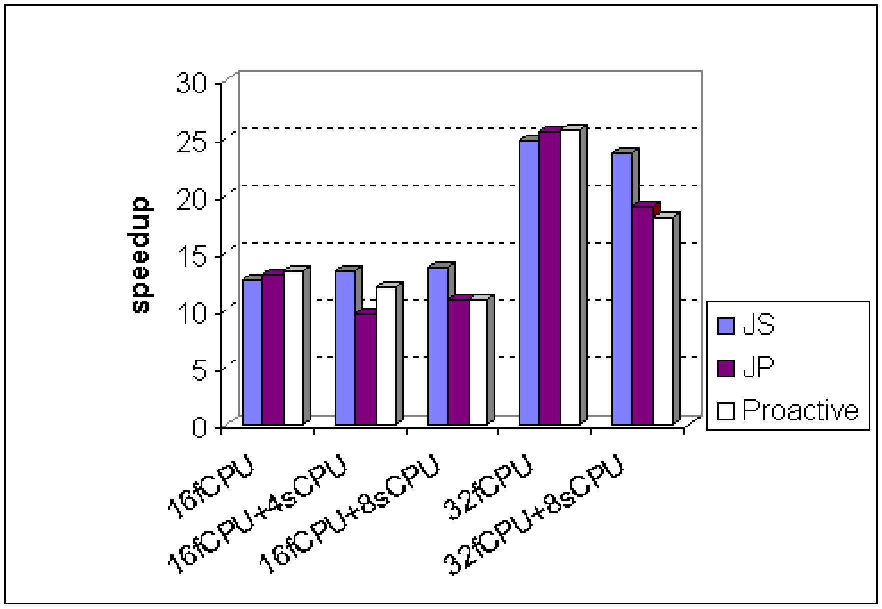


Fig. 7.15. Comparative performance analysis for JavaSymphony, JavaParty, and Proactive DES decryption versions on a heterogeneous cluster of SMP clusters

versions. The performance for 32 fast CPUs and 8 slow CPUs is deteriorating compared to using only 32 fast CPUs, as the additional communication between the two different clusters cannot be compensated with additional computing resources. Nevertheless, even in this case, JavaSymphony achieves better performance than JavaParty and ProActive.

Both JavaParty and ProActive versions achieve similar performance. From this result one may conclude that the performance achieved by JavaParty, which uses a special pre-compiler, can also be obtained with a class library that uses the standard Java compiler, but at the cost of a more complex programming method of ProActive that only partially shields the programmer from low level details.

Overall, we believe that for heterogeneous architectures a system such as JavaSymphony is likely to achieve superior performance compared to semi-automatic systems, as JavaSymphony allows the programmer to control parallelism, locality and dynamic load balancing. Moreover, JavaSymphony shields the programmer from low-level details by supporting a high-level programming paradigm.

7.6 N-body Distributed Application

We have implemented a JavaSymphony application to solve the well-known N-body problem, which determines the motion of a group of interacting particles starting with some initial configuration of positions and velocities in a specified volume of space. Our implementation is based on a sequential code developed in related work [88]. The algorithm is based on a simple approximation: The force on each particle is computed by agglomerating distant particles into groups and using their total mass and center of mass as a single particle. The algorithm consists of three phases, which are repeated in an outermost loop:

1. The volume in which the particles move is subdivided and represented by an octree. Each node in this tree uniquely represents a portion of the volume in which a possibly empty set of particles may reside.
2. Forces aging on each particle are computed through traversing the octree.
3. The velocities and the positions of the N particles are updated by using the new values for the forces.

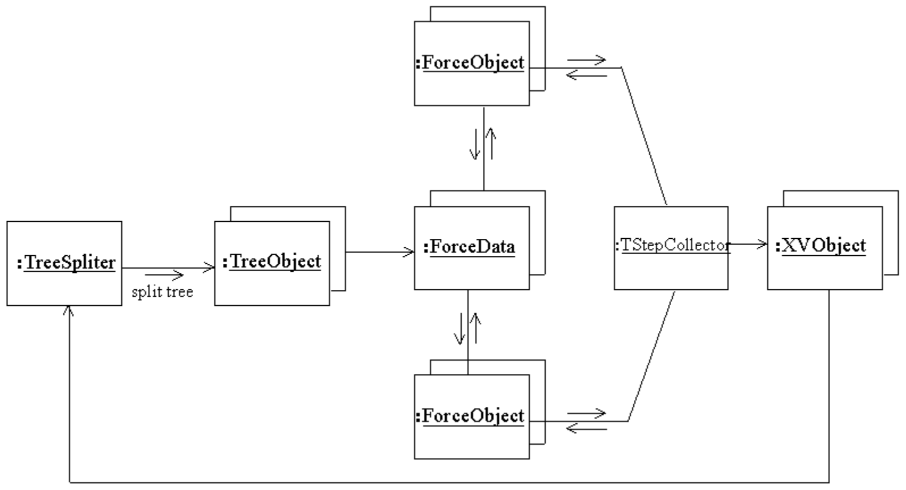


Fig. 7.16. The organization of N-body functional components

The N-body algorithm is divided into several functional components (see Figure 7.16), which are coded as separate JavaSymphony objects. Their functionality and the interaction between them are detailed below:

- One *TreeSplitter* and several *TreeObjects* manage the first phase of the algorithm. The *TreeSplitter* creates the first levels of the octree. The subtrees are distributed to the *TreeObjects*, which build in parallel the full octree.

- Several *ForceData* objects, each being associated with a few *ForceObjects*, manage the second (most computational intensive) phase of the algorithm. The octrees created by the *TreeObjects* are collected by the *ForceData* objects and subsequently processed in parallel by the *ForceObjects*. The *ForceObjects* compute the forces of particles that reside in the space represented by the assigned octree. The *ForceObjects* have direct (local) access to the octree in the *ForceData*. JavaSymphony N-body implementation places at least one *ForceData* object on each SMP node and a number of *ForceObjects* equal to the node's number of CPUs. The *TStepCollector* dynamically controls the workload of the *ForceObjects*. Every time a *ForceObject* becomes idle, the *TStepCollector* assigns it a new set of particles for processing.
- In the final phase, all particles are collected by *TStepCollector* and forwarded to the *XVObjects* which compute the new positions and velocities of all particles. Thereafter, the data is forwarded to the *TreeSplitter*, which starts the next iteration. This phase has not been parallelized, as it requires only reduced computation effort. In order to eliminate extra communication, the *TStepCollector*, *XVObjects* and *TreeSplitter*, which interact with each other, are placed onto the same SMP node.

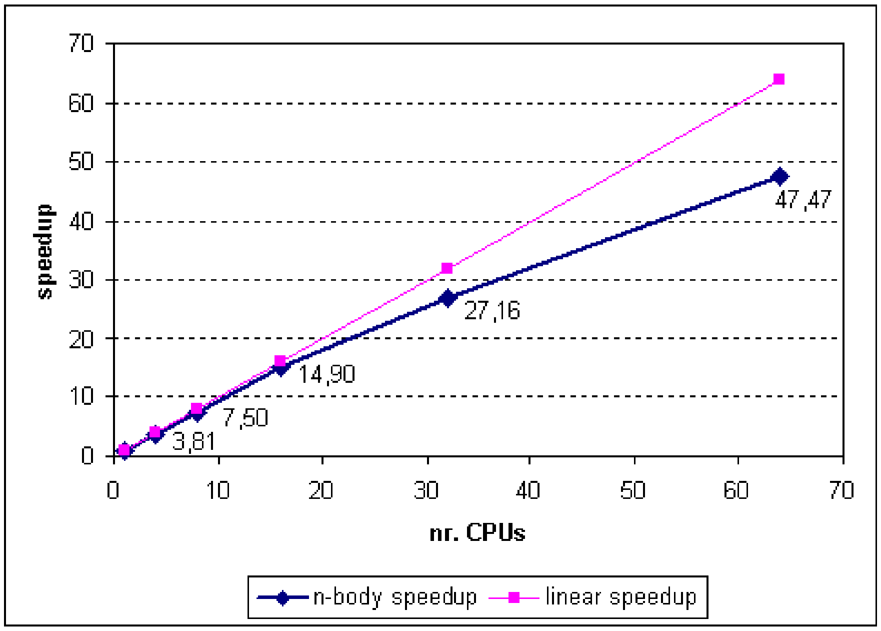


Fig. 7.17. N-body application. Performance result

We have tested our implementation on the previously mentioned cluster [80] with 16 SMP nodes, each of them with 4 Intel Pentium III Xeon 700 MHz CPUs (see Section 7.1.2). Figure 7.17 shows very reasonable speedup values for the JavaSymphony N-body application. The three algorithm phases are executed in a sequential order. Broadcast operations between phases 1 and 2 and barrier synchronisation between phases 2 and 3 influence the general performance of the parallel algorithm. However, phase 2 dominates the computational effort. It can be parallelized almost without communication (except at the beginning and at the end of the phase) by exploiting data parallelism, which explains the good scaling behaviour for small and medium number of CPUs (up to 16 CPUs). For larger number of processors, phases 1 and 3 become more significant, due to the impact of barrier synchronization and broadcast operations.

No. CPUs	Total time	phase 1	phase 2			phase 3
			(all)	(JS overhead)	(comm.)	
4 CPUs	314406	1096	309797.50	12130.00	4606.25	2545
8 CPUs	163361	1174	161496.50	4002.75	1814.13	1261
16 CPUs	81132	1680	77555.94	1875.94	784.75	2611
32 CPUs	44168	2465	38849.75	876.62	528.63	3071
64 CPUs	22143	4026	12971.94	1067.56	506.75	3822

Table 7.4. N-Body timings (in ms)

Table 7.4 shows various timings for this experiment, which includes the overall execution time and time values for all three algorithm phases. Moreover, we were able to isolate the overhead due to JS middleware in phase 2 (see the row named *phase 2 (JS overhead)*). Phase 2 execution time includes computation, middleware overhead and communication. The timings of *phase 2 (all)* correspond to the execution time of the entire phase 2. We can clearly see, that phase 2 dominates the overall performance. The execution times for phases 1 and 3 get bigger for larger processor numbers due to the barrier and synchronization overhead, which prevents linear speedup. The JavaSymphony middleware overhead is very small, compared with the total time for phase 2. The communication overhead, as part of phase 2, decreases for larger cluster size because the data load remains the same, whilst it is distributed to a larger number of ForceObjects.

7.7 Asynchronous Nested Benders Decomposition

The Aurora Financial Management System [89], under development at the University of Vienna, is a decision support system for portfolio and asset liability management (ALM). The classical problem of dynamic asset-liability management seeks an investment strategy, in which an investor chooses a portfolio of various assets, in such a way that some risk-adjusted objective is maximized (for instance the return of investment), subject to the uncertainty of future markets' development, future obligations, and other constraints.

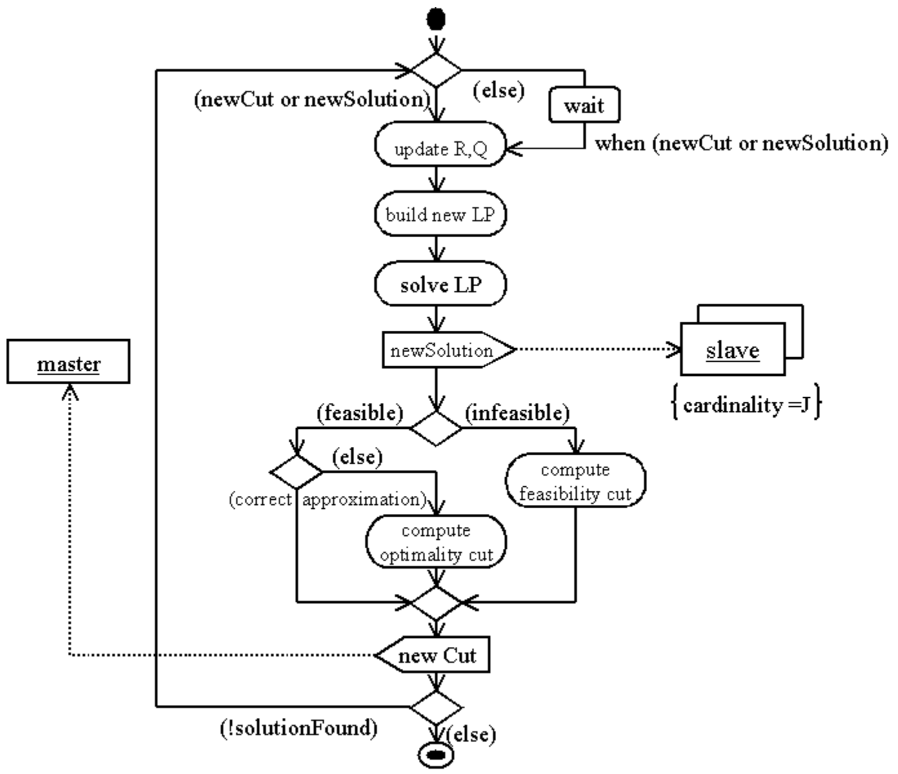


Fig. 7.18. Benders algorithm on each node.

The system contains an optimization module, which solves a stochastic, multistage optimization problem in order to find an optimal investment plan. Realistic models for the future development of asset prices taking into account many risk factors result in very large scenario trees.

Among other methods [90], the optimization module makes use of decomposition techniques [91], which allow for solving large-scale tree structured optimization problems very efficiently. In addition, due to their inherent parallelism, they are well suited for parallel implementation [92] .

The entire optimization problem is decomposed into a set of local problems (Fig. 7.18), corresponding to the nodes of the scenario tree. Every tree node represents an active object (executed by a single thread). This contributes to the solution of the entire problem in an iterative manner:

- The node solves its local problem by means of a Linear Program (LP) solver;
- The local problem is updated with information from neighbour nodes (i.e. the predecessor and the successors in the scenario tree), and then the node solves the updated problem etc.

An asynchronous parallel version of the nested Benders decomposition method has been implemented on top of different parallel programming platforms [93, 94]. The JavaSymphony version has been built starting from existing Java code, which contains an algorithm layer calling asynchronous communication operations implemented in an underlying coordination layer. Only the coordination layer and the initialization part have been adapted to the JavaSymphony API. The threads associated with the tree nodes are encapsulated within JS objects; the algorithm code remains unchanged (Fig. 7.18).

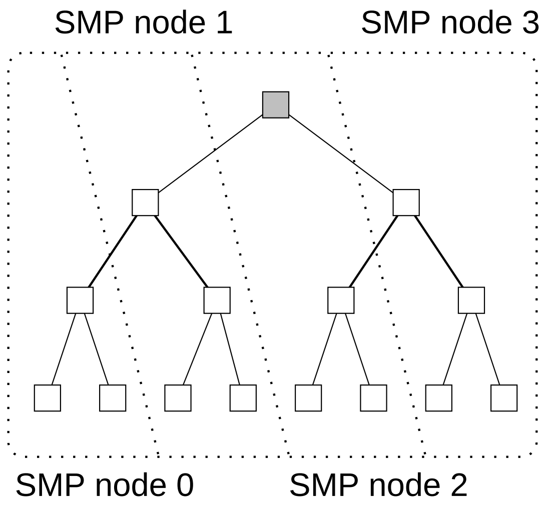


Fig. 7.19. Load balancing mapping strategy for a binary tree with 15 algorithm nodes, on a cluster with 4 SMP nodes

We use an optimal load balancing strategy for mapping the tree nodes onto SMP computing nodes (Fig. 7.19). Communication among tree node objects residing on the same compute node is performed through direct access. JavaSymphony remote method invocation is used for communication between objects residing on different computing nodes.

We have tested the application on binary scenario trees of different sizes. The node objects are working on test input data reflecting realistic optimization problems. The experiments have been performed onto SMP nodes (each comprising 4 CPUs) of a dedicated Beowulf cluster (see Section 7.1.2), used in most of the previous experiments. The speedup results for up to 8 SMP computing nodes are displayed in Fig. 7.20.

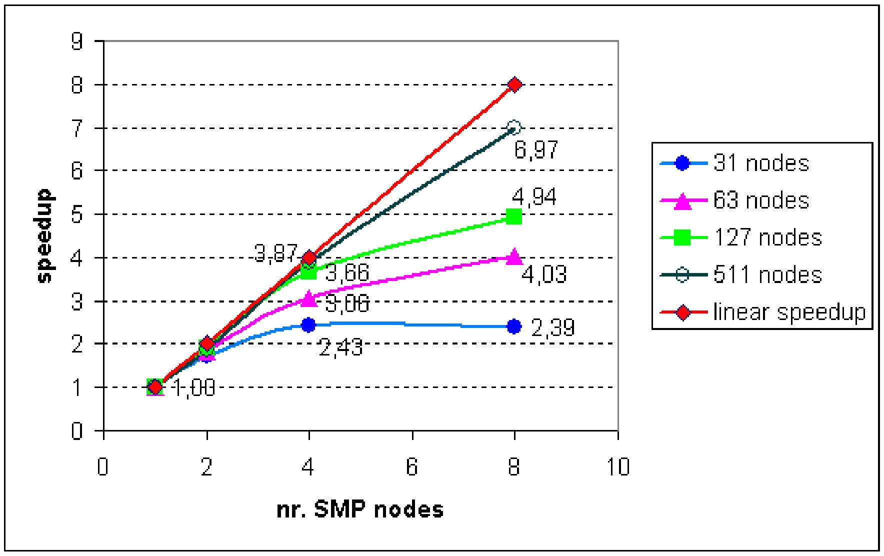


Fig. 7.20. Speedup for Benders decomposition

Some of our measurements demonstrate that the effective time for computation (spent on the CPU) is small in comparison to the actual execution time. The latter value is strongly influenced by synchronization between nodes and idle times due to specific criteria waiting to be met, and by the large number of tree node objects per compute node, each having its own thread of execution. For the tree with 31 nodes, the SMP nodes are used inefficiently because of the small number of tree nodes per each CPU and the reduced computation load for each tree node. The speedup improves with the number of tree nodes, because the CPUs are used more efficiently. The best performance is achieved for 511 tree nodes. Therefore, we believe that our JavaSymphony-based implementation of the nested Benders decomposition is suitable for realistic optimization problems (in terms of tree and local problem sizes).

7.8 Performance of Micro Benchmarks

In order to evaluate the middleware overhead, we have developed a set of micro benchmarks to test several of the basic elements of the JavaSymphony constructs: remote object creation, method invocations and object migration.

All the experiments have been conducted by using 2 or 3 SMP nodes of the cluster described in the previous experimental sections. Each of the benchmarks uses two types of objects: a *Tester* and a *Tested* object. The *Tester* creates one or more *Tested* objects and performs specific benchmark operations on them. Three metrics are analyzed:

- **Total time** is the total elapsed time for the execution of a particular benchmark;
- **Average time** is an estimation of the time it takes for a particular activity to complete;
- **Rate** defines an estimation of the number of these activities per second. The time values are computed in milliseconds.

We have examined the overheads for creating local and remote JS objects, for the remote method invocations: synchronous, asynchronous and one-sided, respectively for object migration. The results are analyzed in the followings.

	CL			CR		
No. obj.	Total time	Avg. Time	Rate	Total time	Avg. Time	Rate
1	38.00	38.00	26.32	69.50	69.50	14.39
10	229.60	22.96	43.55	252.40	25.24	39.62
100	1622.00	16.22	61.65	2165.00	21.65	46.19
1000	14305.00	14.31	69.91	20537.20	20.54	48.69
10000	140546.00	14.05	71.15	149317.33	14.93	66.97

Table 7.5. Creation of JSObjects - remote and local.

CR and CL - Object Creation Benchmarks

These benchmarks are used to study the overhead due to the creation of JS objects, locally for the CL benchmark or remotely for the CR benchmark. The *Tester* creates a variable number of *Tested* objects placed on the same level-1 VA for the CL benchmark, respectively on a distinct level-1 VA for the CR benchmark. The number of *Tested* objects varies from 1 to 10^4. The results are presented in Table 7.5.

As expected, remote creation adds the cost of communication to the generic cost of the object creation. At the same time, by increasing the number of operations the performance increases to a peak value: The *rate* of creations is slowly growing. We can see that for a large number of operations, the values for CR and CL become relatively close to each other.

SI, AI, OI - Method Invocation Benchmarks

For each type of remote method invocation (synchronous, asynchronous or one-sided), a micro benchmark is used to estimate its overhead (i.e. SI, AI, OI benchmarks). For these benchmarks, the *Tester*, builds one single *Tested* object onto the same SMP node, respectively onto a remote one and invokes one of its methods for a number of times.

	SI			OI		
Nr.oper	Total time	Avg Time	Rate	Total time	Avg.Time	Rate
Local						
1	17.67	17.67	56.60	18.67	18.67	53.57
10	209.67	20.97	47.69	181.67	18.17	55.05
100	1444.67	14.45	69.22	1458.00	14.58	68.59
1000	11823.33	11.82	84.58	11197.00	11.20	89.31
10000	116495.00	11.65	85.84	110731.67	11.07	90.31
Remote						
1	18.33	18.33	54.55	20.00	20.00	50.00
10	246.00	24.60	40.65	166.67	16.67	60.00
100	2289.33	22.89	43.68	1780.33	17.80	56.17
1000	14634.00	14.63	68.33	11286.00	11.29	88.61
10000	116459.00	11.65	85.87	102819.33	10.28	97.26

Table 7.6. Variable number of method invocations for SI, OI.

	AI			
Nr.oper	Total time	with results	Avg Time	Rate
Local				
1	17.67	18.67	17.67	56.60
10	179.67	181.00	17.97	55.66
100	1422.33	1423.33	14.22	70.31
1000	11099.67	11101.00	11.10	90.09
10000	108095.00	108096.33	10.81	92.51
Remote				
1	20.67	21.67	20.67	48.39
10	206.00	207.00	20.60	48.54
100	1670.00	1671.00	16.70	59.88
1000	11578.33	11579.67	11.58	86.37
10000	103229.00	103230.00	10.32	96.87

Table 7.7. Variable number of method invocations for AI

For SI the communication is bi-directional, while for AI and OI this is uni-directional. Additional time is necessary for AI-benchmark to receive the results. Since the invoked method is empty (exits immediately), we assume that the actual execution time is negligible. The tables 7.6 and 7.7 show the performance values for the SI and OI, respectively for AI benchmarks.

MIG - Migration Benchmark

For the MIG benchmark, we test the overhead for migrating objects. One *Tested* object is created and we invoke n times an empty method (with no computation overhead) in two different ways: with or without object migration.

	No migration			With migration		
Nr.Oper.	Total time	Avg Time	Rate (op/sec)	Total time	Avg Time	Rate (op/sec)
1	19.60	19.60	51.02	242.20	242.20	4.13
10	199.20	19.92	50.20	1959.00	195.90	5.10
100	1563.20	15.63	63.97	13597.40	135.97	7.35
1000	12557.00	12.56	79.64	122129.20	122.13	8.19
10000	115643.20	11.56	86.47	1228014.67	122.80	8.14

Table 7.8. Variable number of method invocations/migrations for MIG.

In the first case, the *Tested* object is migrated before each method invocation. The migration overhead is the difference between the two executions. Table 7.8 shows that the overhead is significant. This is caused by the complex migration protocol, which involves a lot of additional communication when methods for migrated objects are invoked. The results demonstrate that intensive migration is very expansive and suggest that it should be used only in special cases, when the performance improvement generated is significant and compensates the cost of migration itself.

7.9 Scheduling Workflow Applications in JavaSymphony

In order to demonstrate the usefulness of JavaSymphony workflow scheduling technique, we have implemented several real world JavaSymphony workflow applications.

The original workflow applications comprise a set of scripts and executables that run on distributed computing resources. The JavaSymphony graphical tool for workflow composition has been used to build the applications' workflow. Java wrapper classes are used to run the components of the original applications onto distributed resources, as activities of the workflow. Files are transferred between activities by using the JavaSymphony Runtime System.

We have enhanced the JavaSymphony enactment engine to export the workflow application as input file for Condor DAGMan [37]. For each finished workflow activity, a Condor job submission file is created as well. Condor DAGMan does not offer support for sequential/parallel loops and conditional branches. Therefore, these elements of the workflow are not present in the output DAG, which comprises only the activities that have finished their execution, and the associated control dependencies.

We compare the scheduling performance of the two schedulers: JavaSymphony dynamically builds a schedule for the workflow application, whilst Condor DAGMan uses the static DAG, as built by JavaSymphony workflow enactment engine. Consequently, the two schedulers execute the same sets of activities, restricted by the same control dependencies.

The activities are executed on the same set of computing resources, in a Condor pool of workstations (Table 7.9). The workstations are heterogeneous,

Machine	JavaMFlops
ankogel.dps.uibk.ac.at	34.866943
blindis.dps.uibk.ac.at	38.801613
ganot.dps.uibk.ac.at	27.678907
gramul.dps.uibk.ac.at	39.018559
mulle.dps.uibk.ac.at	34.130119
ochsner.dps.uibk.ac.at	58.082180
olperer.dps.uibk.ac.at	38.438427
petzeck.dps.uibk.ac.at	70.885361
pleisen.dps.uibk.ac.at	33.361057
quirl.dps.uibk.ac.at	37.276459
schareck.dps.uibk.ac.at	34.449875

Table 7.9. Condor pool of workstations ranked by JavaMFlops attribute

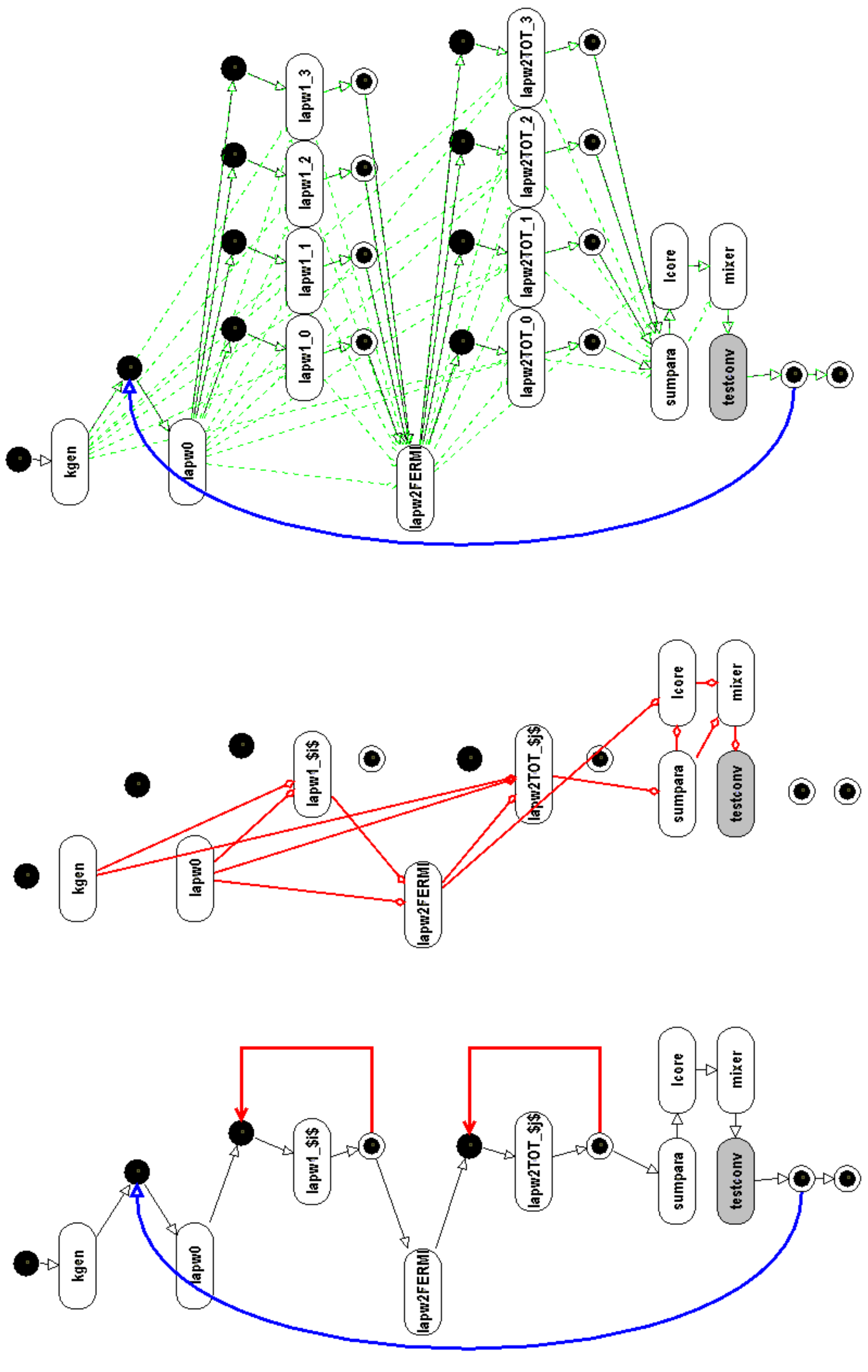
ranked according to *JavaMFlops* attribute of Condor machine ClassAd, which determines the speed of the machine by using the SciMark2 benchmark [95], at the time Condor is started onto the machine. We have used artificially controlled execution times for the workflow activities: Each activity *act* is associated with a computing cost $cost(act)$ expressed in FLOPs. Consequently, the execution time of one activity *act* on a machine m is determined as $cost(act)/power(m)$, where $power(m)$ is the value of *JavaMFlops* attribute associated with the machine. Therefore, one activity finishes in shorter time if it is mapped onto the machines that are higher ranked. On the other hand, the execution of a specific workflow activity takes the same amount of time, if mapped on the same machine, no matter if the activity is executed by using the Condor or the JavaSymphony scheduler.

JavaSymphony determines the schedule of the workflow application based on the activity execution times onto the resources. The activities are mapped as Java objects onto the workstations. The Condor DAGMan scheduler uses the DAG file to schedule the DAG of the workflow application. Each activity is associated with a Condor Java-universe job submission file, which executes the very same Java class onto the remote resources. The *JavaMFlops* is used as a priority associated with the resources, so that the stronger machines are preferred over the weaker. However, Condor cannot use the estimated execution times to make scheduling decisions.

7.9.1 Wien2k

WIEN2k [96] is a program package for performing structure calculations of solids by using density functional theory, based on the full-potential (linearised) augmented plane-wave ((L)APW) and local orbitals (lo) method.

The components of the WIEN2k package can be organized as a workflow (Fig. 7.21). The *lapw1* and *lapw2_TOT* tasks can be solved in parallel by a fixed number of so-called *k-points*. This is modelled by two parallel loops in



(a) Control flow (b) Data dependencies (c) Parallel loop elimination

Fig. 7.21. Wien2k workflow

the workflow graph. Without the parallel loops, the workflow graph becomes quite complex (Fig. 7.21(c)). Various files are sent from one workflow activity to another, which determine complex data dependencies between the activities (Fig. 7.21(b)). At the end of the main sequence of the activities, a dummy activity *testconv* performs a convergence test to determine if the calculation needs to be repeated. This is modelled by the main sequential loop.

We have successfully built a JavaSymphony workflow application on top of the WIEN2k package. We used HEFT (Heterogeneous Earliest Finish Time) [47] list scheduling algorithm combined with the dynamic scheduling strategy described in Fig. 5.23, to schedule and run this application on a set of workstations.

Figure 7.22 presents the schedules for the two executions of the Wien2k workflow: by using the JavaSymphony scheduler, respectively the Condor DAGMan scheduler. The experimental run uses 8 *k-points* and the calculation within the main loop is repeated 3 times. This gives us a number of 67 activities in the execution plan, for both JavaSymphony and Condor DAGMan. As we can see in Fig. 7.22, both schedulers prefer to use the better machines, as ranked by *JavaMFlops* attribute. Since the width of the graph (the maximum number of activities that may run in parallel) is 8, the slowest machines (i.e. pleisen and mulle) are not even used. Moreover, JavaSymphony scheduler decides not to use two other slow machines (i.e. quirl and ankogel), based on the estimations of the execution times for each activity. On the other hand, Condor DAGMan uses the next best available resource whenever a new activity is ready to run. As we can see in this case, using more resources to run the workflow application does not necessarily improve the performance for DAGMan scheduler, and JavaSymphony scheduler outperforms DAGMan scheduler by a factor of 1.64.

7.9.2 Invmod/Wasim

The IWI Institute uses the hydrological model WaSiM-ETH [97] for the simulation of catchment's water balances. WaSiM-ETH is an extensive state of the art program, which allows the user to choose different modules for the simulation of the single hydrological processes. Depending on the available data and the simulation scope, various sub-models may be chosen. WaSiM-ETH is a raster-based model working on a regularly spaced grid. For each grid, a water balance is calculated by using meteorological data and derived spatial data.

In order to calibrate the model and to evaluate the model's efficiency, time series of observed discharges at river gauges are required. If discharge measurements for a sufficient time frame exist, an Inverse Modelling approach allows a fast and more objective estimation of simulation parameters. The Inverse Modelling software InvMod calibrates WaSiM-ETH parameters by using a Levenberg-Marquardt- algorithm to minimize the least squares of the differences of the measured and the simulated runoff for a determined period.

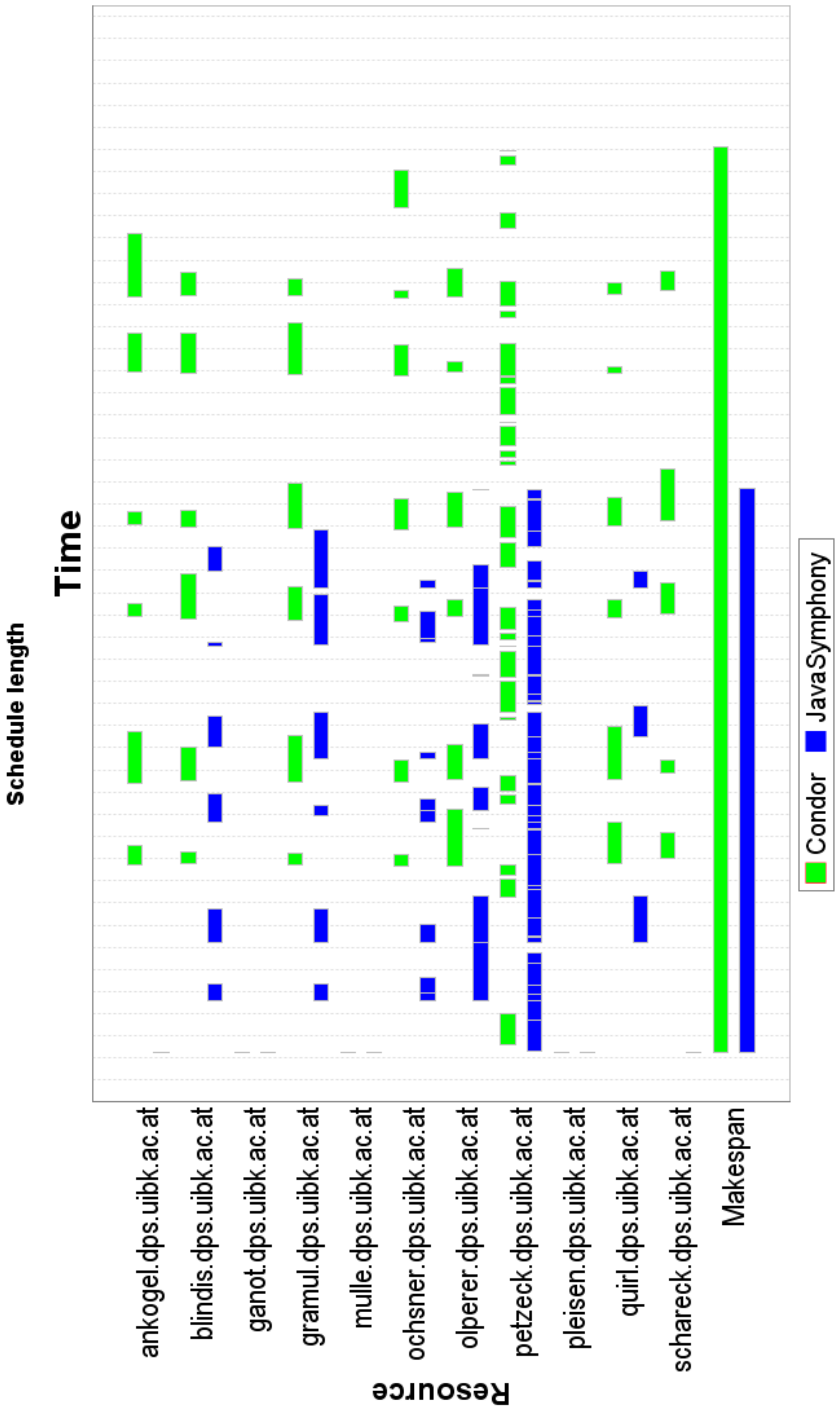
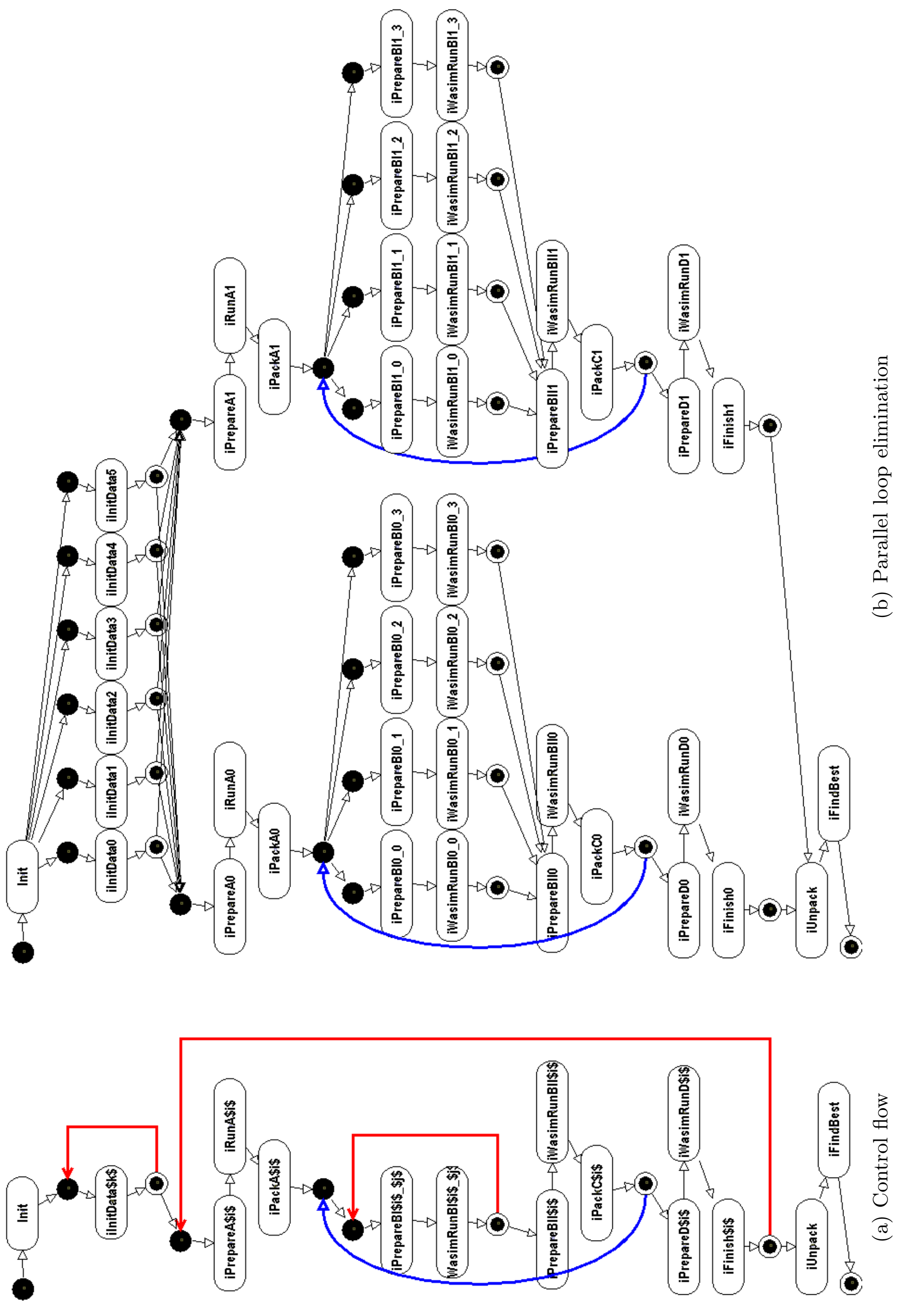


Fig. 7.22. Gannt Chart for Wien2k workflow execution. JavaSymphony vs. Condor



(a) Control flow

(b) Parallel loop elimination

Fig. 7.23. Invmod/Wasim workflow

At the University of Innsbruck, as a cooperation between the Institute of Hydraulic Engineering and the Institute of Computer Science, a Grid parallel version [98] of the program has been developed, based on the Java CoG libraries. Based on this study, we have implemented a JavaSymphony workflow application for Wasim/Invmod (Fig. 7.23).

The structure of the workflow graph is quite complex. The application starts with an initialization phase, which prepares several input data sets to be distributed onto the computing resources: activities *Init* and *iInitData*. The main computation may be split in several identical runs, which is modelled by the external parallel loop. For our experiment, we have chosen two identical runs. Within the external parallel loop, the Invmod computation is split into three sequential parts:

1. *Wasim1* is sequential and comprises the activities *iPrepareA*, *iRunA*, *iPackA*;
2. *Wasim 2-3* is the main part of the original Invmod version and runs in parallel according with a *parameters number* input value; At the end of the phase, *iWasimRunBII* merges the results of the parallel runs. The phase is repeated in a sequential loop until a termination condition evaluated by *iPackC* activity is fulfilled. The experiment runs two iterations of the sequential loop and uses the *parameters number* with value 4 (i.e. 4 iterations of the internal parallel loop).
3. *Wasim 4* comprises a sequence of several other activities.

At the end of the workflow execution, the results are merged. The execution plan of the experiment comprises 65 activities. Figure 7.24 presents the schedules for the two executions of the Invmod/Wasim workflow: by using the JavaSymphony scheduler, respectively the Condor DAGMan scheduler, as described above. For the same reasons as for the Wien2K experiment (Section 7.9.1), JavaSymphony scheduler outperforms DAGMan scheduler by a factor of 1.22. Both schedulers prefer the better machines. Since the width of the DAG is 8, some of the slowest machines are not even used. Condor DAGMan uses more of them, but this does not help it to improve the performance in comparison with JavaSymphony counterpart.

7.9.3 Montage

Montage [99] is a software system for generating astronomical image mosaics according to user-specified size, rotation, WCS-compliant projection and coordinate system, with background modelling and rectification capabilities (Fig. 7.25).

Based on the Montage tutorial [99] and on the study done by Truong et al. [100], we have built a JavaSymphony workflow for Montage application. In Fig. 7.26, the activities *mProject(i)* are used to reproject input images to a common spatial scale. A *mAdd* activity is used to co-add the reprojected images. Both types of activities access build image tables created by a

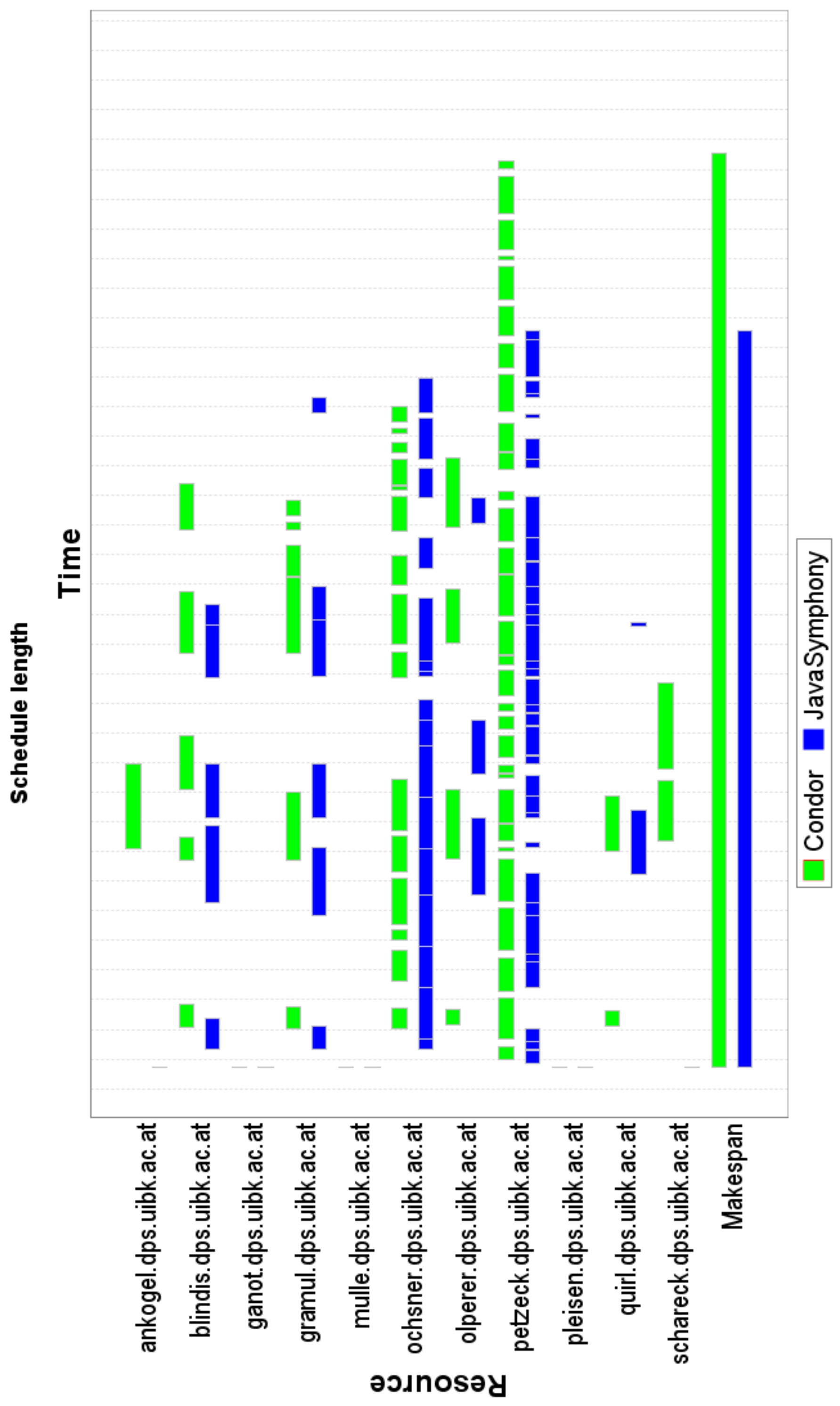


Fig. 7.24. Gannt Chart for Invmod/Wasim workflow execution.

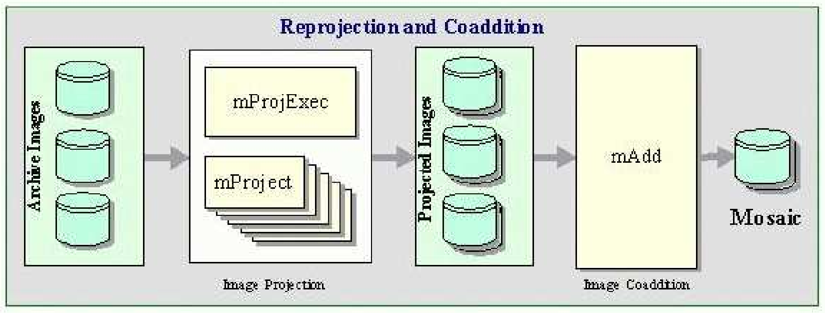


Fig. 7.25. MONTAGE. Process flow overview (Montage Tutorial)

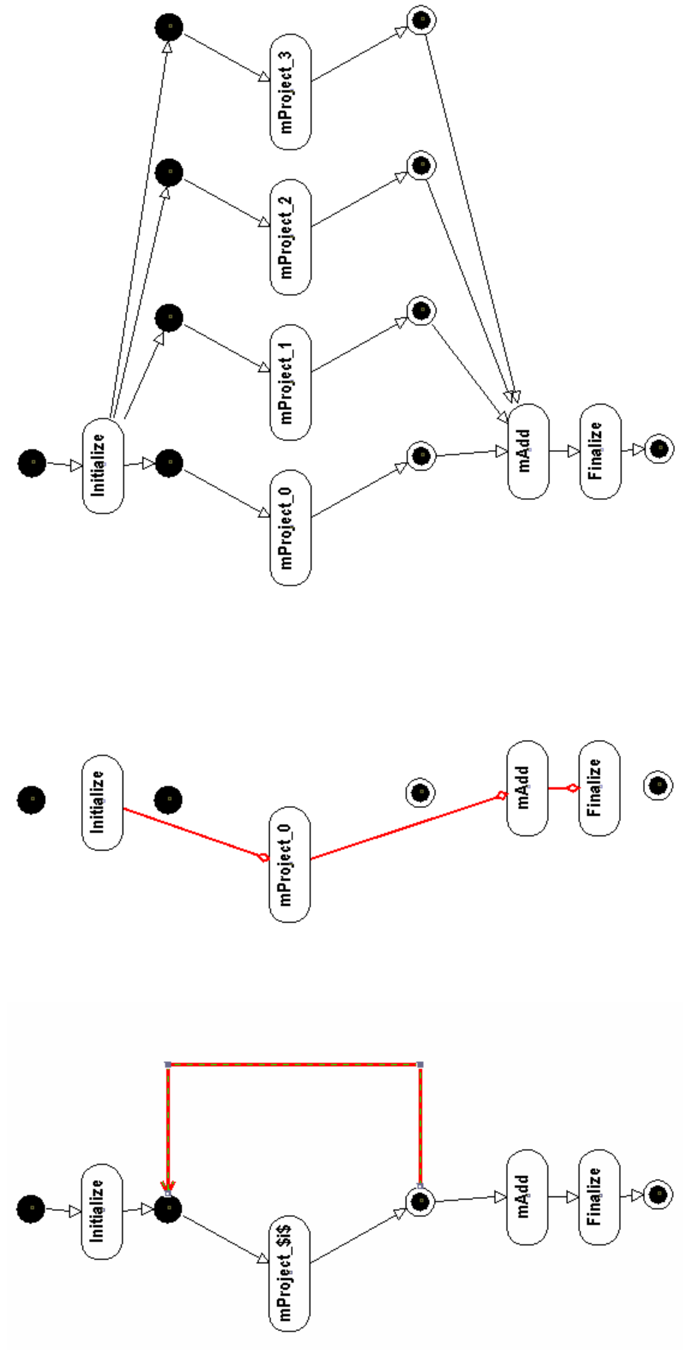
call to *mImgtbl* module. We consider the calls to this module as part of the *mProject(i)*, respectively *mAdd* activities. *Initialize* activity distributes archive images, whilst *Finalize* collects the resulted mosaics. The data transfers between the activities are modelled by data links (Fig. 7.26(b)): Raw images are transferred from user site to computing sites, projected images produced by *mProject* are transferred to the site where *mAdd* is placed, and finally the result mosaics are collected at the user site. The *mProject* activities work in a fork-join fashion, which is modelled by a parallel loop (Fig. 7.26(c)), with the number of iterations specified at design time.

Figure 7.27 presents the schedules for the two executions of the Montage workflow: by using the JavaSymphony scheduler, respectively the Condor DAGMan scheduler. The workflow graph for the Montage application is simpler than in the previous experiments (Sections 7.9.1 and 7.9.2). Since there are no branches or sequential loops, JavaSymphony does not need to reschedule the workflow application. Therefore, a statically computed schedule is used. We use a number of 64 parallel runs of the *mProject* activity, which gives a total of 67 activities in the execution plan. Due to the large width of the graph, the resource utilization is balanced. However, DAGMan scheduler introduces gaps in the utilization of the best machine (i.e. petzeck) due to synchronization problems, which is the main reason for performance deterioration. In this experiment, JavaSymphony scheduler outperforms Condor's DAGMan by a factor of 1.15.

7.10 Summary

In this chapter, we have extensively tested the suitability of the JavaSymphony for a variety of distributed and parallel algorithms. Several JavaSymphony applications have been tested on both homogeneous and heterogeneous distributed architectures.

Moreover:



(a) Control flow (b) Data dependencies (c) Parallel loop elimination

Fig. 7.26. Montage workflow

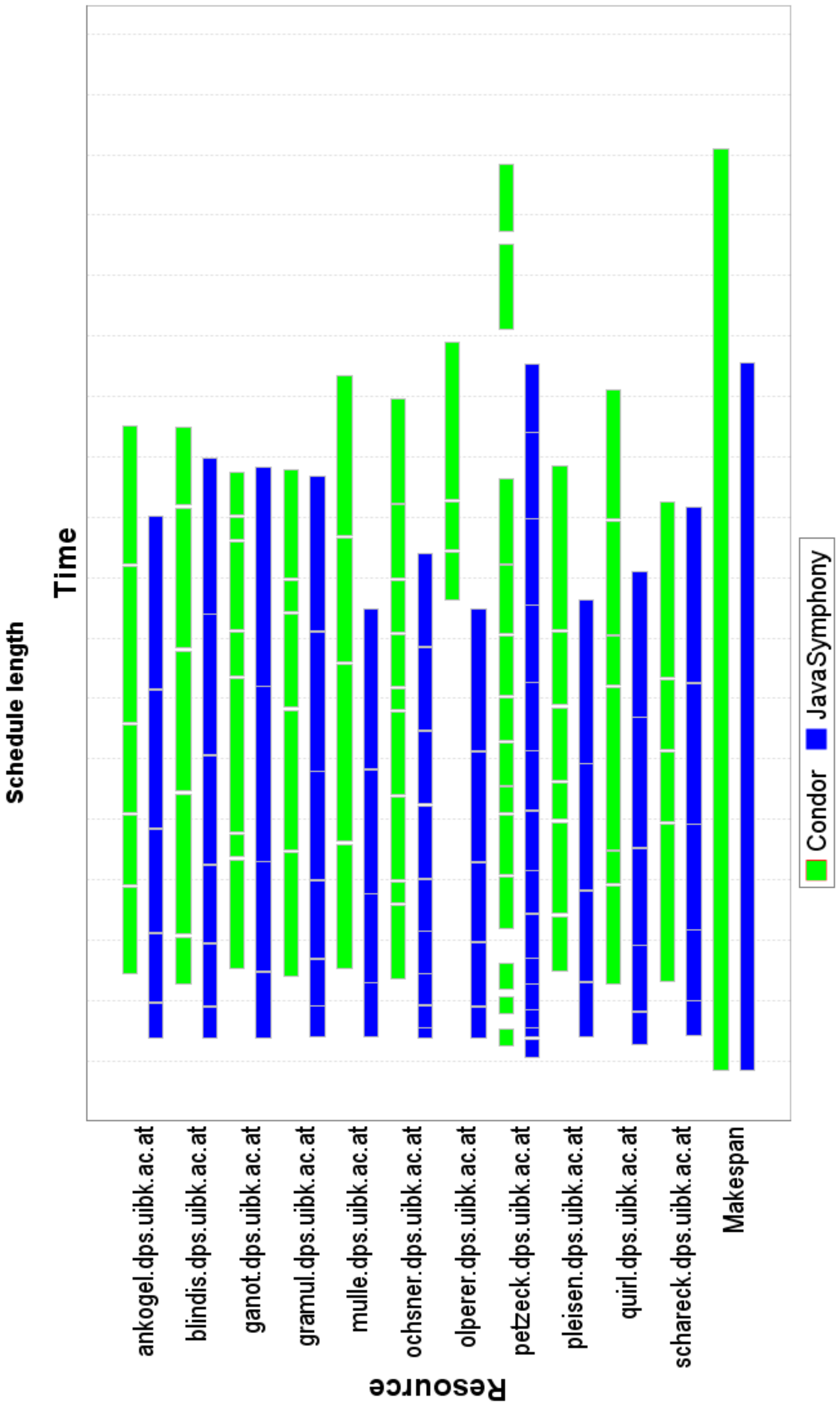


Fig. 7.27. Gannt Chart for Montage workflow execution.

- We have demonstrated several high level features of the JavaSymphony programming paradigm, including distributed virtual architectures for managing heterogeneous distributed physical architectures, distributed events, and distributed synchronization barrier;
- We have investigated the overhead due to our middleware. For this purpose, we have analyzed several real-life applications and a set of special designed micro benchmarks to test the basic features of JavaSymphony programming paradigm (e.g. remote object creation, remote method invocations and migration). We have proved that the JavaSymphony middleware overhead is acceptably small, which makes our software suitable for medium- to coarse-grained parallelism.
- We have compared JavaSymphony with two other Java-based programming paradigms for concurrent and distributed applications, namely JavaParty and ProActive. The results show that a system such as JavaSymphony is likely to achieve superior performance for heterogeneous architectures, compared to semi-automatic systems.
- We have demonstrated the JavaSymphony workflow scheduling technique. The comparison with the well-known Condor DAGMan shows the superiority of using a dynamic scheduling strategy.

Related Work

This thesis has tackled problems regarding Java-based distributed programming, workflow model, workflow scheduling, and resource brokering. The most relevant related work in each of these areas will be discussed in separate sections of this chapter.

8.1 Java-based Distributed Systems

There is a large amount of related work, which makes collaborative use of computational resources over a global network, including low-level communication systems (e.g. MPI [15] and PVM [101]) or higher-level dedicated systems (e.g. Globus [12], Legion [102], and NetSolve [103]). Although these systems offer heterogeneous collaboration of multiple systems in parallel – some of them in wide-area setting – they involve rather complex maintenance of different binary code, multiple execution environments, etc. CORBA [19] defines a middleware that bridges distributed objects across heterogeneous environments. It allows client objects to invoke server objects across the network. All objects expose a well-defined interface in the Interface Definition Language (IDL) and thus can be invoked from anywhere in the network. CORBA as well as Globus and Legion can be used to build the JavaSymphony Runtime System (JRS). However, we have decided to use Java/RMI instead, assuming that it entails less complexity and overhead.

Jini [104] provides a sophisticated technology for interconnecting generic devices that provide services to other devices or users. Devices and their services make themselves public by registering with a lookup service. Once connections to devices are made, the lookup service is no longer involved in the interactions between clients and services. Jini could be used to build part of the JRS. However, whereas the JRS is currently built on a thin protocol layer to provide JavaSymphony functionality (such as providing VAs), we believe that performance problems may arise by using Jini, due to larger protocol overheads.

		AdJava	Aglets	Aleph	Ajents	Bayanihan	Charlotte	Frontier	JADA	JavaParty	JavaSpaces	JavaSymphony	Javelin/CX	JAWS	Ninflet	POPCORN	ProActive	Voyager
Arhitecture	one-dimensional	X	X	X	X				X	X	X			X			X	X
	multi-dimensional					X	X	X				X	X		X	X		
	dynamic reconfiguration											X						
System information												X		X	X			?
Selective code loading					X		X					X					X	X
Load balancing	static									X		X						
	dynamic					X	X	X					X	X	X	X		
Communication	synchronous	X	X		X					X		X		X			X	X
	asynchronous		X		X							X					X	X
	one side											X					X	X
	shared-space			X					X		X							X
	client-server only					X	X	X					X	X	X	X		
Remote object	single-threaded	X	X	X					X	X		X	X	X	X		X	X
	multi-threaded				X							X						?
	dynamic conversion											X						
Locality control				X	X	X					X	X					X	
Migration	explicit	?	X		X					X		X			?		X	
	automatic	X				?	X	?					?	X	X	?		X
Transaction mechanism			X	X	X						X				X		X	
Synchronization	lock/unlock			X								X						
	barrier			X								X					X	
	shared-space			X					X		X							
	others																X	X
Events			X	X							X	X						X
Fault tolerance					X	X	X	X					X		X	X	X	

Table 8.1. Properties of related Java-based middleware

In order to overcome system complexity, several research groups have introduced Java-based global computing systems that benefit from Java's platform independence. These efforts can be broadly classified into two categories.

The first category concentrates on improving the implementation of the Java Virtual Machine (JVM) [105, 106]. With a runtime system written in C, Manta [107] provides a C-runtime system that uses a native compiler to translate from Java directly to executable code. This system, however, imposes changes to the Java syntax and semantics. Other work focuses on improving the performance of Java object serialization [108], or Java/RMI (e.g. KarMI [109], NRMI [110], or NexusRMI [111]). JavaSymphony could directly benefit from these optimizations, as the JRS runs on any standard compliant JVM. Several other projects into this category, such as MultiJav [112], cJVM [107], Jackal [113], Java/DSM [114], JESSICA [115] (and its optimized successor JESSICA2 [116]), or Jikes RVM [117] (formerly known as Jalapeno from IBM), modify the JVM and adapt it to a distributed shared memory model.

The systems in the second category try to alleviate the usage of Java as a distributed programming language. There are two ways of realizing this thing:

- By extending Java with special distribution constructs and semantics (e.g. JavaParty and Charlotte), which also require changes to the Java compiler and/or JVM. Furthermore, several projects extend Java language with support for popular lower-level programming model, such as data-parallel model (e.g. HPJava [118], Timber [119] or Titanium [120]), PVM (e.g. JavaPVM [121], JPVM [122], or Jcluster [123]) and MPI (e.g. javaMPI [124] or Jcluster).
- By offering special class libraries, which are compatible with any JVM. JavaSymphony falls into this sub-category, by providing a Java library that simplifies the development of distributed Java applications.

Table 8.1 shows a few alphabetically ordered projects, that fall in the second category, along with a list of supported features. The projects address different aspects of distributed computing and, therefore, some of the features do not apply to all of them.

- **Architecture.** According to the organization of the computing resources, we classify the systems in: one-dimensional systems, when the resources are organized as a simple list of machines, and multi-dimensional systems, which use structures that are more complex. The systems that are oriented to web volunteer computing (e.g. Bayanihan [125], Charlotte [126], POPCORN [127]), or systems which employ a three-tier architecture (e.g. Javelin [128], Ninflet [129]) are included in the second category; however, the resource organization is transparent to the user or application. For JavaSymphony, the allocation and organization of resources is explicit for each application. In addition, JavaSymphony can dynamically change the configuration of the resources at runtime.
- **System information.** A few systems provide an interface to runtime information about the computing resources used by the applications.
- **Load balancing.** Several systems provide automatic load balancing, which can be static, if the resources are allocated in the initialization phase, or dynamic, if there is the possibility to reallocate the computing resources at runtime.
- **Selective code loading.** This feature is valid if the code for distributed applications could be selectively loaded onto the distributed computing resources.
- **Communication.** We identify a few types of communication which are supported by related middlewares: Synchronous, asynchronous, one-sided

communication are supported for remote method invocation. Share-spaced communication is characteristic to systems that use virtual shared memory or shared objects to coordinate remote threads. Client-server only communication applies to web-computing based systems for which the client applications use a set of server/hosts that do not communicate with each other.

- **Remote Objects.** This concept may have distinct meanings for distinct systems. Some systems use remote distributed objects for which several methods can be executed at the same time (multi-threaded). For others, only a single method can be executed at a time or even the object is in fact a remote thread (e.g. Aleph [130]). For task-oriented systems, this feature does not apply. In some cases the users are allowed to dynamically convert local objects to remote objects.
- **Migration.** Objects can be migrated among computing resources under certain conditions (e.g. load balancing strategies, locality). Some systems support migration among computing resources. Migration can be explicit, by using library calls inside the code, or automatic, when the system (based on an internal status) decides to migrate objects/tasks.
- **Transaction mechanism.** Some systems can interrupt the execution of a thread of execution, save the current state and resume execution or rollback. This is in particular useful for safe migration of objects that are changing their state.
- **Synchronization.** Some systems provide synchronization mechanism like lock/unlock, barrier, or based on shared space (shared objects in a virtual shared memory).
- **Events.** A few systems offer an event mechanism, which allows asynchronous interaction among parallel threads of execution.
- **Fault tolerance.** Fault tolerance mechanisms are provided to permit recovery when resources become unavailable. Systems oriented to web computing are in particular interested in this feature.

We extensively studied the available documentation for the systems in Table 8.1. In some cases, in which it was not clear if a specific feature is supported, we have marked this in our table with "?".

Charlotte [126] supports a distributed shared memory on top of the JVM, but does not enable the programmer to control locality of data. Instead, programs supported by Charlotte alternate sequential and parallel phases and define routines for parallel execution. Charlotte also changes the semantics of Java language and requires special JVM/compiler to run the applications.

POPCORN [127] introduces a market-based mechanism of trade in CPU time, which motivates processors to provide their CPU cycles for other people's computations.

Bayanihan [125] supplies a framework for volunteer computing based on Java applets. Hosts over Internet volunteer to do work by visiting a web page and executing an applet, while client applications provide independent tasks to be computed. Automatic dynamic scheduling and fault tolerance mechanisms are provided, but none or little support is provided for communication or synchronization among hosts. A similar approach is implemented by Frontier [131], a commercial distributed application from Parabon. The client applications solve tasks distributed as Java class files.

Javelin [128] (and its successor CX [132]), Ninflet [129], SuperWeb [133], and JaWS [134] employ a three-tier architecture with broker, client and host entities. Clients seek computing resources by submitting their work in form of applets or Java objects, register with a broker and submit their work in the form of an applet/Java object. Hosts donate resources, contact the broker and run applets. These approaches are appropriate for master/worker and divide-and-conquer applications, but lack flexible mechanisms for communication among hosts. They also provide very little help to control locality. Ninflet and JAWS offer a migration mechanism. JAWS documentation speaks about possible communication among the hosts, but the mechanism is not clear. JAWS also uses a credit-based mechanism to allocate computing resources.

Jada [135] and JavaSpaces [136] can be considered as Linda derivatives which provide none or limited support to control locality. Parallel threads of execution can coordinate their activity or communicate, by using a virtual shared space.

Aglets [137] is a mobile agent system in which agents have a relative independence and can migrate or communicate via messages or events. Typically, agent systems do not provide references to remote objects, which restricts the development of flexible distributed applications.

Aleph [130] extends the thread parallelism by using remote threads that are distributed among several (remote) JVMs. The remote threads communicate via *GlobalObjects*, which provide a virtual shared space.

Voyager [138] is a Java-based commercial middleware with several characteristics similar to JavaSymphony. Voyager supports several messaging types, remote references, several widely-used messaging protocols, complex resource utilization mechanism, and security.

JavaParty [139, 86], ProActive [140] and Ajents [141] present more similarities with our middleware. These systems support the creation of remote objects, which are distributed among computing resources, and communication among these objects.

JavaParty extends Java with a class modifier *remote*. Objects generated for remote classes are distributed among several computing resources. A disadvantage is that a special (pre)compiler is required to execute the application. JavaParty also uses optimized serialization and Java/RMI [142].

AdJava [143] adopts an approach similar to JavaParty: It uses a *Distribute* special keyword to indicate objects that can be distributed, and a pre-compiler to produce Java-compatible code from applications that use this extension.

ProActive (formerly known as Java//) adds a lot of functionality (polymorphism, future objects, sophisticated synchronization libraries) at the cost of increased complexity. The RMI details are not sufficiently hidden and the programming effort becomes considerable. The functionality of the software has been recently enhanced with new features to support peer-to-peer, data-parallelism, several Grid/cluster protocols for communication, fault-tolerance and checkpointing [87].

Ajents has influenced JavaSymphony's programming model for remote object creation, asynchronous remote method invocation and class loading. However, Ajents, just as most other systems, does not allow the programmer to explicitly control object locality. Ajents also does not support multi-dimensional architectures, one-sided remote method invocations, selective class-loading to specific computing nodes, and access to hardware/software system parameters as JavaSymphony. Ajents, however, offers a sophisticated check-pointing mechanism and supports the migration of objects while their methods are executed.

8.2 Algorithms for Scheduling Tasks-based Applications in Distributed Systems

Most of the related work assumes that parallel applications comprise a set of tasks that work together to realize the goal of the main application. A task, as part of a parallel application is usually supposed to be a computation unit that runs sequentially on a single computing resource. The goal of the scheduling algorithm is to find a mapping of the tasks of one or several parallel applications onto the computing resources such that a performance metric (e.g. execution time, throughput, communication cost, computation cost) is optimized (i.e. minimized or maximized). In most of the cases, the scheduling problem is NP-complete [21], and the algorithms are heuristics that provide a close-to-optimal solution.

The problem of scheduling multiple tasks on distributed systems was extensively studied and it still represents an important research problem. There are plenty of algorithms introduced to solve this problem. However, each algorithm is based on some preliminary assumptions (e.g regarding application model or target computing architecture), and tries to optimize a specific performance metric. These characteristics may significantly vary from one algorithm to another, which makes the classification of scheduling algorithms and the comparison of their performance rather difficult.

Detailed comparisons are presented in several papers [49, 46, 42]. However, none of such overviews could cover all the existing scheduling heuristics. We propose a simple classification of the scheduling algorithms for task-based parallel applications, according to two main criteria: the type of the application and the type of the target system.

The Application Type

There are two main classes of task-based parallel applications:

- **Meta-tasks** comprise tasks that are executed independently.
- **Applications based on inter-dependent tasks** assume that there is a dependency relation over the set of the comprising tasks (usually based on data communication).

Therefore, we identify two types of scheduling algorithms: scheduling algorithms for meta-tasks, respectively scheduling algorithms for precedence-constrained task graphs. The algorithms in the second category commonly have higher complexities in comparison with those in the first category, due to the more complex structure of the application, and commonly address only Directed Acyclic Graphs (DAGs) of tasks.

We discuss more details and present several examples for these two types of algorithms further in this section.

The Target System Type

From this point of view, there are scheduling algorithms that assume a homogeneous system and algorithms for heterogeneous systems. Generally, the algorithms for heterogeneous systems have a higher complexity than those for homogeneous systems.

Scheduling heuristics for homogeneous systems.

Heuristics of this type have been used for the compilation of parallel applications, to improve their performance on specific target machines. In this case, it is assumed that a parallel application consists of sequential tasks, and that the target computing architecture consists of several identical processors. Each of the tasks is assigned to a processor. Mainly because homogeneous architectures were commonly used for parallel computing before the heterogeneous systems, this class of heuristics has been intensively studied. There are several distinct assumptions and/or scheduling techniques, which further divide these algorithms in sub-classes:

- **Heuristics for unbounded number of processors** include DSC [144], EZ [145], LC [146]. The assumption that the number of processor is unlimited is not applicable in most cases. The algorithms of this type employ a technique called *clustering*, which groups the tasks in an unbound number of clusters, in order to reduce the completion time. A post-processing step is needed to map the clusters onto actual processors.
- **Scheduling heuristics that use duplication** for a bounded number of processor [46, 49] include DSH [45], BTDH [50], CPFD [51], PY [52], LWB [53], LCTD [54], MJD [55], DFRN [56]. Duplicating task usually results in better scheduling performance at the cost of significantly higher complexity. The reason for using duplication is to reduce the communication cost of some schedule, by duplicating the tasks that intensively communicate with other application tasks.

- **Scheduling heuristics without duplication** for a bounded number of processors. Most algorithms in this class are based on the *list scheduling* technique. The basic idea is to assign priorities to the tasks of a parallel application, and to place the tasks in a list in descending order of priorities. The tasks with higher priority are examined for scheduling before those with a lower priority. The large number of list scheduling algorithms is the result of the numerous variations in the method of assigning priorities and maintaining the list of ready tasks. The algorithms of this type are known to perform well at relatively low cost.
  Radulescu et al. [49] further divide the list scheduling algorithms in two classes:
  – List scheduling with *static* priorities (eg. MCP [57], DPS [58], HFLET [59], CPND [60], CPM [61]). For these algorithms, the priorities associated with the tasks are computed at the beginning. Thereafter, each task is selected according to its priority and it is scheduled on the "best" processor (according to specific criteria).
  – List scheduling with *dynamic priorities* (e.g. ETF [62], ERT [63], DLS [66], DCP [64]). In this case, the priorities are computed gradually at each iterative step, taking into consideration the tasks already scheduled. Note that, even if the priorities associated with the tasks are dynamically computed, the algorithms are considered to be static (i.e. the schedule does not change at runtime).

All the algorithms mentioned above assume a **DAG-based application structure**. We could not identify algorithms dedicated for scheduling **meta-tasks on homogeneous systems**. However, the work of Hagerup et al. [147] outlines several scheduling strategies to allocate chunks of tasks onto a parallel machine (i.e. homogeneous system). These include: Static-chunking (SC) [148], self-scheduling (SS), Guided self-scheduling (GSS) [149], Trapezoid self-scheduling (TSS) [150], Factoring [148], TAPER [151], and Bold scheduling [147]. In this case, the basic idea is that the idle processors access a central monitor, which assigns a batch of previously unassigned tasks to it. The above-mentioned scheduling strategies assume that task execution times follow a probabilistic distribution known a priori. Under these assumptions, the scheduling algorithm tries to determine the optimal size of a batch, in order to minimize the overall finishing time.

A distinct approach is used by James et al. [152]. This paper focuses on cluster computing, which may include also heterogeneous systems, but it describes results obtained on a homogeneous system. Several scheduling algorithms/strategies are described: **round-robin** scheduling, **clustered round-robin** scheduling, **minimal adaptive** scheduling, **continual adaptive** scheduling, **first-come first-served (FCFS)** scheduling. These could be used by the so-called queuing software (e.g. PBS [153], LoadLeveler [154], Condor [155]), to place independent jobs onto processing nodes, both for homogeneous and heterogeneous systems. An important advantage of these scheduling strategies is that they do not require estimations of job execution times.

Scheduling heuristics for heterogeneous systems.

There is a growing interest for building parallel applications, which target heterogeneous systems. A new class of scheduling algorithms is needed for the applications that run in heterogeneous environments. The main difference is that the tasks of an application have distinct behaviours when mapped onto specific computing elements of a heterogeneous system. The scheduling algorithms have to consider the machines' heterogeneity, which makes them more complex than similar algorithms for homogeneous systems.

There are many heuristics algorithms for scheduling DAGs of tasks onto heterogeneous systems, including LDBS [65], DLS [66], GDL [66], BIL [67], Hybrid Re-mapper [68], MH [69, 47], LMT [70, 47], TDS-1 [71], TDS-2 [72], FCP [73], FLB [73], HEFT [47], HCNF [74], CPOP [47]. Many of these algorithms use the well-known list scheduling technique, as for homogeneous systems.

On the other hand, many research groups focused on scheduling (large) sets of independent tasks (meta-tasks) on heterogeneous systems. This class of algorithms is better represented than in the homogeneous case. Heterogeneous environments like network of workstations or computational Grids seem to be more suitable for the so-called embarrassing-parallel applications, which can be decomposed into a (large) number of independent sub-tasks.

Several heuristics for scheduling meta-tasks on heterogeneous computing system are presented in [42]. These include:

- **OLB - Opportunistic Load Balancing** [156, 157] assigns each task in arbitrary order to the next available machine.
- **UDA: User-Directed Assignment** [156, 157] (also known as LBA - Limited Best Assignment) assigns each task in arbitrary order to the machine with the best expected execution time.
- **Min-Min** [156, 157, 21] assigns at each step a task that has the minimal value of the overall minimum completion time.
- **Max-Min** [156, 157, 21] assigns at each step a task that has the maximal value of the overall minimum completion time.
- **Sufferage algorithm** [44] chooses first a task with the highest sufferage value(i.e. the difference between its best and second best completion time).
- **Genetic algorithms (GA) [158, 159], Simulated Annealing (SA) [160, 159] and Genetic Simulated Annealing (GSA) [161, 159]** use a representation of the schedule as a chromosome and try to find the chromosome with the best fitness value (the makespan of the schedule). These algorithms iteratively use random generated solutions to improve the fitness value, until some specific stopping criteria are met.

- **Tabu search** [162, 159] also uses random generated solutions to search in a large solution space. A *tabu list* is used to keep track of the regions of the solution space in order to avoid searching for them again.
- A^* **heuristic** [163, 159] performs an extensive tree-based search in large solution spaces. It provides better solutions at the cost of significantly larger scheduling overhead.

8.3 Workflow Research in Distributed and Grid Computing

8.3.1 Workflow Languages

Building DAG-based application workflows is an old topic, which has been widely addressed in numerous scientific fields. Following the recent Grid hype, workflow applications have become very popular in Grid community too, and many research and industry groups have proposed language standards to model and develop workflow applications.

Web services [164] have become the base technology for developing secure and reliable Grid services, and consequently workflow languages for Web services have faced significant development in recent times. Web Services Flow Language (WSFL) [165], currently developed by IBM, describes the composition of Web services by combining a flow model and a global model. The first one defines the workflow activities, interconnected through control and data links, which correspond to the composite Web services, respectively the order in which the activities are executed. The global model defines how the activities are mapped into operations of the individual Web services. The Grid Service Flow Language (GSFL) [166] is a proposed adaptation to the OGSI-based Grid services model.

Microsoft has proposed XLANG language [167] to model business processes as autonomous agents.

The Business Process Execution Language for Web Services (BPEL4WS) [168] unifies two previously competing standards: WSFL and XLANG. It provides a language for the formal specification of the business processes and business interaction protocols by extending the Web Services interaction model to support business transactions.

Gridbus [169, 170] provides a simple XML-based workflow language to define tasks and their dependencies. The workflow description language of Gridbus is aimed towards enabling the expression of parameterization and users' QoS requirements.

GridAnt [171] re-uses the Ant [172] framework to develop a simple, yet powerful client side workflow system for Grids. GridAnt is a tool used not only to map complex client-side workflows, but also as a simplistic client to test the functionality of different Grid services. Applications with simple process flows can benefit from GridAnt without having to endure any complex workflow architectures.

The above-mentioned languages are based on the widely used XML (eXtensible Markup Language) [173]. On the other hand, Condor DAGMan (Directed Acyclic Graph Manager) [174] uses a proprietary format to specify the DAG-based workflows. An input script describes the DAG, whilst each node in the graph is associated with a Condor submit description file.

AGWL (Abstract Grid Workflow Language) [175], developed at University of Innsbruck within the Askalon project [176], provides an advanced and user-oriented Grid workflow language, which shields the complexity of the underlying Grid infrastructure and runtime environment from the application developer. AGWL aims to address several commonly drawbacks of the existing work: control flow limitation (e.g. DAG-based workflows lack branches and loops), limited mechanism for expressing parallelism, restricted data flow mechanism (e.g. restricted to sending/receiving files), implementation specifics and low level constructs, etc. The high level AGWL program is compiled to a CGWL (Concrete Grid Workflow Language), which is managed by a WfMS.

We do not intend to compete with highly complex workflow definition languages. Instead, the JavaSymphony specific XML-based specification language for workflow applications is simple, in order to allow an easy manipulation of the workflow structure by a scheduler.

8.3.2 Workflow Representation

Graph-based modeling [177] of workflows has been in the attention of workflow research groups as well. Graphical representation of the workflows is very intuitive and can be easily handled by non-expert users.

UML Activity Diagrams [178, 41] or Petri Nets [179] have been extensively studied as alternatives for the representation of the workflows ([180, 39]).

Petri Nets [179] are directed graphs with specific properties that can model sequential, parallel, repetitive and conditional execution of tasks. Petri Nets have been used in workflow management systems such as GridFlow [181], FlowManager [182], and XRL/Flower [183]. However, Eshuis et al. [180] consider that Petri Nets need semantic extensions in order to model workflow activities accurately. UML activity diagrams do not have these problems.

Van der Aalyst et al. [39] describe in detail a large set of workflow patterns, which are represented in UML activity diagram model. UML activity diagrams have been extended and are commonly used to represent workflows in related work [41, 184]. We use ourselves a graphical representation based on the UML activity diagram. Complex workflow specification languages or complex workflow patterns are not commonly associated with advanced scheduling techniques for distributed workflow applications. Therefore, we prefer to use a reduced set of workflow patterns [39], in order to be able to investigate such advanced scheduling techniques.

Besides Petri Nets, and UML activity diagrams, other workflow management system may use their own graphical representation of workflow components. For example, Triana [185, 186] allows users to utilize predefined software components and assemble them into DAG-based workflows.

8.3.3 Workflow Management Systems

The research in the workflow applications field does not confine itself to workflow languages or workflow representation model. Research groups also address enactment engines, scheduling and fault tolerance for workflow applications. In the following, we describe how several workflow management systems address some of these issues.

The XCAT [187] Application Factories address Grid workflow applications within the Common Component Architecture (CCA). XCAT allows components to be connected to each other dynamically. However, scheduling of components and fault tolerance are not addressed.

Within the myGrid toolkit [188], the IT Innovation Workflow Enactment Engine [189] is a workflow orchestration tool for Web services. It supports an XML workflow definition language that is based on WSFL and supports control and data flow.

The collaboration of IT Innovation, EBI and other members of myGrid project has produced SCUFL (Simple Conceptual Unified Flow Language) used within Taverna [190, 191] workflow management system. Taverna supports DAG-based workflows and its own graphical representation of the workflows. User-controlled fault tolerance is achieved by setting up various specific parameters such as the number of retries, time delay, alternate processor or critical level for faults on each processor. Scheduling is not addressed: each task is individually allocated at the time it is ready to run, according to a just-in-time planning scheme.

The Unicore project [192, 193] introduces a DAG-based graphical model, which supports batch jobs that run over a set of distributed resources and temporal dependencies between them. Recently, advanced flow controls including conditional execution (e.g. if-then-else) and repeated execution (e.g. do-n, do-repeat) have been added. Fault tolerance and scheduling are not addressed: Unicore assumes that the workflow runs on reliable resources and uses just-in-time planning to run the jobs.

Pegasus (Plan for Execution in Grids) [194] is a workflow manager for the Chimera project [195], in the broader context of GryPhyN (Grid Physics Network) [196]. There are two stages in the development of a workflow application: first, an abstract workflow is built. The workflow jobs are interconnected through data dependencies only. In the second stage, the abstract workflow is mapped into a concrete one, which maps each graph node into a corresponding resource. The resource selection is based on random allocation or on performance prediction. The authors advocate artificial intelligence planning [197]. However, the mapping is static, and the workflow is transformed into a script for Condor DAGMan and associated Condor jobs. Pegasus does not address fault tolerance.

DAGMan [174] submits jobs to Condor in an order that obeys the DAG dependencies, and processes the results. The jobs are scheduled only when they are ready to run, based on Condor specific techniques such as resource matchmaking and cycle stealing. In case of failure, a rescue DAG is created, which can be used to re-submit the failed jobs.

GridFlow [181] is more concerned about service-level scheduling and workflow management, and less about workflow specification. GridFlow is built on top of ARMS [198], an agent-based resource management system for Grid computing. A workflow in GridFlow represents a flow of activities, each representing a sub-workflow that is executed in a local grid. GridFlow comprises a user portal and services for both global grid workflow scheduling, built on top of ARMS, and for local grid sub-workflow scheduling. At each local grid, sub-workflow scheduling and conflict management are processed on top of existing performance prediction based task scheduling system.

GrADS (Grid Application Development Software) [43] provides application-level scheduling to map the tasks of workflow and meta-task applications onto a set of distributed resources. The research within GrADS project focuses on developing new scheduling and rescheduling methods. Min-min, Max-min and Sufferage heuristics are used to schedule DAGs of tasks. The scheduler obtains resource information by interrogating grid services such as MDS (Monitoring and Discovery Service) [199] or NWS (Network Weather Service) [200]. The scheduling strategy estimates the performance of a workflow component on a single Grid node, based on parameters such as resource usage, number of floating-point operations or memory access patterns. Furthermore, GrADS approach monitors resource performance and the agreement between the application demands and resource capabilities (performance contract). Based on these, corrective actions can be taken, according to two approaches: stop/restart implies migration of task and data; process swapping implies the use of idle backup machines.

On the other hand, most systems for allocating tasks on grids, (e.g. DAGMan [174], Pegasus [201]), currently allocate each task individually at the time it is ready to run, without aiming to globally optimise the workflow schedule. In addition, they assume that workflow applications have a static DAG-based graph, which may be seen as a too restrictive constraint.

The DAG scheduling problem has been intensively studied in the past, mostly in connection with parallel application compiling techniques. A parallel application is represented by a DAG in which nodes represent application tasks (computation) and edges represent inter-task data dependencies (communication). Numerous scheduling techniques and scheduling heuristics have been developed for both homogeneous and heterogeneous systems, including list scheduling [47, 46, 74, 73], scheduling with task duplication [45, 50, 51, 56, 46], or clustering technique [144, 145, 46]. However, these heuristics assume a static application graph and they statically compute the schedule before the execution is started. Static scheduling of static DAG structures is, however, too restrictive for the new generation of Grid workflow applications. Therefore, we propose a new approach that includes loops and conditional branches into the workflow model and extends the static scheduling with novel dynamic scheduling techniques to accommodate these new constructs.

8.4 Resource Management Tools

Resource brokering is a fundamental component that addresses the discovering resources for the consumers purposes. Often, the term gets mixed up with closely related concepts such as resource discovery/dissemination, resource monitoring, or resource management. The last concept generally covers a wider area of research, which may include scheduling as well [202]. In this section, we investigate several resource management tools for Grid computing, focusing mainly on their resource brokering features.

Collecting present or predicted information about the available resources is essential for any resource management system. Globus Toolkit [12] offers Grid Monitoring and Discovery Service (MDS) [199], a powerful tool for resource management, which is able to provide dynamic information regarding available services or machine specifics. Moreover, Network Weather Service (NWS) [200] produces forecasts for the performance of network and computing resources. A prototype of NWS was implemented for Globus. However, a resource broker in the Globus system is still missing.

Numerous resource management systems make use of a well-known matchmaking mechanism, which is best illustrated by Condor [155]. Condor scheduler is based on the matchmaking between the *ClassAdds* [203] for the resources and those for the Condor jobs. These include resource properties, respectively the job placement constraints, similar to the resource attributes and activity constraints described in Chapter 6.

GrADS [43] resource selection framework [204], addresses the discovery and configuration of physical resources that match application requirements. It provides a declarative language, using set matching techniques, which extends Condor matchmaking.

Legion [102] is in essence an operating system for Grid. The scheduling process in Legion broadly translates to placing objects on processors. In Legion, a *collection* object (similar in spirit to MDS) functions as a database for system information. Application-specific information can be specified in a *class* object, which includes the selection of a specific scheduler.

CrossGrid [205] uses a matchmaking scheduling mechanism, based on allocating resources to tasks expressed in JDL (Job Description Language). The scheduling is not separated from the resource brokerage and both are addressed by the same service called *Resource Selector*.

Nimrod-G [206] is a Grid resource broker for managing and steering task farming applications, which follows a computational market-based model for resource management. Nimrod-G uses the services of other resource management systems, such as Globus or Legion, for resource discovery and resource dissemination.

AppLeS [207] is another example of Grid-level resource management system that uses the services of other systems such as NetSolve [103], Globus or Legion for task execution. Moreover, AppLeS project focuses on developing scheduling agents for individual application templates. The AppLeS scheduling methodology makes extensive use of NWS facilities.

GridARM [208], part of Askalon project [176] is a Grid resource management system that provides automatic resource brokerage, *Virtual Organization*-wide fine-grained authorization, advanced reservation and negotiation between a potential client and resource provider. Further author's work [209] proposes an ontology-based resource description, discovery and correlation mechanism, for an automatic brokerage service of the Grid resource management system.

The work of Krauter et al [210], which includes a taxonomy and a survey of resource management systems for distributed computing, can be consulted for further details about the related work in this area.

Our theoretical framework presented in Chapter 6 is complementary to the work described above, by modelling resource attributes and activities requirements for a generic matchmaking mechanism, by proposing a model for advance reservations, or by addressing the resource monitoring problems. The model applies well to JavaSymphony Runtime System and to JavaSymphony scheduler/enactment engine for workflow applications. However, we believe that its applicability can be easily extended to other related systems.

9 Conclusions

We have developed JavaSymphony, a new high-level programming paradigm for Java-based parallel and distributed applications. In this thesis, we have presented JavaSymphony high-level features, which can be used to implement distributed applications for wide classes of heterogeneous systems, ranging from small-scale cluster computing to large scale Grid computing. Furthermore, we have implemented the JavaSymphony Runtime System, a middleware to support the execution of distributed applications that have been developed by using the JavaSymphony programming paradigm. We have enhanced this middleware with features for the development and management of distributed workflow applications, and we have introduced novel scheduling strategies to improve the performance of such applications. The usefulness of our programming paradigm is demonstrated through various experiments.

This chapter concludes the thesis with a summary of our main contributions and an outline of the potential future research directions.

9.1 Contributions

In this section, we outline the main contributions of this thesis.

9.1.1 The JavaSymphony Programming Paradigm

JavaSymphony programming paradigm, on the one hand, supports automatic mapping, load balancing and migration of objects without involving the programmer. On the other hand, in order to enhance the performance of distributed applications, JavaSymphony provides a semi-automatic mode, which leaves the error-prone, tedious, and time consuming low-level details (e.g. creating and handling of remote proxies for Java/RMI) to the underlying system, whereas the programmer controls the most important strategic decisions at a very high level.

We have introduced JavaSymphony programming paradigm with various features highly useful for the implementation of distributed applications:

- **Dynamic virtual architectures.** The JavaSymphony dynamic architectures (VAs) are used to manage distributed and heterogeneous resources at a high-level. The VAs support a lock/unlock control mechanism, which enables safe concurrent access. The locality of the resources is controlled by an allocation algorithm which optimally maps resources at the creation of the VAs, so that they are as closed to each other as possible, according to a configuration set up by the JavaSymphony Administration Shell (Sections 4.2, 4.3).
- **System parameters and constraints.** The VAs allow access to static and dynamic information regarding the hardware capabilities. JavaSymphony constraints can be used to examine the properties of physical resources, which include static parameters such as machine name, operating system and peak performance parameters, or dynamic parameters such as system load, idle times, and available memory. This information can be inspected at runtime, thus allowing the user to take critical decisions regarding resource management and object mapping/migration, in order to dynamically enhance the overall performance of the distributed application.
- **Flexible distributed objects.** JavaSymphony objects (JS objects) enable the use of various types of remote method invocations: synchronous, asynchronous or one-sided. Concurrent access to one object can be managed by using a lock/unlock mechanism. JS objects may be migrated to other computing resources automatically or based on user instructions. Generic (Java) objects can be transformed into JS objects and may be accessed by remote entities. Concurrent method calls for one object can be done in parallel for multi-threaded objects, or sequentially ordered for single-threaded objects. This behaviour can be changed dynamically. Moreover, JavaSymphony provides facilities to make JS objects persistent by saving and loading them to/from external storage.
- **Distributed synchronization.** The programming paradigm supports the synchronization of parallel and distributed threads of executions. On the one hand, synchronization of parallel asynchronous method invocations allows flexible coordination of multiple threads at the initiation site. On the other hand, a distributed barrier mechanism allows non-centralized coordination of concurrent threads.
- **Distributed event mechanism** is flexible and supports asynchronous communication between remote entities within a distributed application. Any two objects can become the producer, respectively the consumer of a specific event. Moreover, events can be filtered according to event source, event target, producer location, consumer location, or event type.

9.1.2 The JavaSymphony Runtime System

In order to support the JavaSymphony programming paradigm and its high-level programming constructs, we have designed and implemented the JavaSymphony Runtime System. The JavaSymphony Runtime System is a distributed middleware, whose components (called agents) are running in background onto each machine that participates to the execution of JavaSymphony applications. Moreover, the JavaSymphony Runtime System supports monitoring and management of the distributed resources and the applications, enables communication based on Java RMI mechanism among application's objects, and a simple fault-tolerance mechanism.

We have formally described the JavaSymphony Runtime System by using a variant of Pi-calculus, a calculus for mobile processes. A formal description of the middleware is implementation-independent and thus useful for possible porting it onto other platforms (e.g. Grid or Web services instead of Java agents and SOAP-based communication instead of Java RMI). Moreover, this description supports in-depth theoretical analysis and verification.

9.1.3 Scheduling Techniques for Distributed Workflow Applications

The thesis proposes a novel method for dynamic scheduling of workflow applications in heterogeneous environments. The method can be used with many of the existing scheduling algorithms (Section 8.2). In order to use the scheduling method in the context of JavaSymphony, we have developed:

- A workflow model, which extends the classical DAG-based workflows with loops and conditional branches elements. The novel workflow model allows dynamic changes of the execution graph based on data available at runtime. Furthermore, we introduce a formal representation of the workflow applications, based on Pi-calculus, which allows theoretical understanding of the interactions between the workflow elements.
- A XML-based specification language used to describe the workflow and its element, according to the workflow model. The language is simple, yet expressive and allows specifying QoS information associated with the resources and with the workflow activities.
- An user interface for building the graphical representation of the workflow. The graphical elements map directly to the elements of the workflow model. The interface uses the workflow specification language to save/load the workflow to/from external storage.
- An enactment engine, part of the user interface. The enactment engine uses our scheduling strategy to schedule workflow applications onto a set of heterogeneous resources. A JavaSymphony application is automatically built and executed, according to the workflow specification, respectively the scheduling strategy.

We analyze our scheduling method for a few other scheduling objective functions. Our analysis includes an economical cost model and a scenario with multiple objective functions.

Moreover, we propose a theoretical approach to model the resources and QoS associated with the workflow activities in relation with the resource broker.

9.1.4 Published Contributions

The work described has been widely accepted as important contribution in the field of programming paradigms for distributed and Grid computing, respectively scheduling for distributed applications. In the following, we present the list of papers published or accepted for publication, which have contributed in these areas.

Journals

- Thomas Fahringer and Alexandru Jugravu. *JavaSymphony: A new programming paradigm to control and to synchronize locality, parallelism, and load balancing for parallel and distributed computing.* in Concurrency and Computation. Practice and Experience. 17(7-8):1005-1025, June/July 2005.
- Alexandru Jugravu and Thomas Fahringer. *JavaSymphony, a Programming Model for the Grid.* in Journal for Future Generation Computer Systems - Grid Computing: Theory, Methods and Applications , Promotional Issue January 2005

Referred Conferences

- Thomas Fahringer, Alexandru Jugravu, Beniamino Di Martino, Salvatore Venticinque, Hans Moritsch. *On the Evaluation of JavaSymphony for Cluster Applications.* in Proceedings of the IEEE International Conference on Cluster Computing CLUSTER2002, Chicago, Illinois, Sept. 2002.
- Thomas Fahringer and Alexandru Jugravu. *JavaSymphony: New Directives to Control and Synchronize Locality, Parallelism, and Load Balancing for Cluster and GRID-Computing.* in Proceedings of Joint ACM Java Grande - ISCOPE 2002 Conference, Seattle, Washington, Nov. 2002.
- Alexandru Jugravu and Thomas Fahringer. *On the Implementation of JavaSymphony.* in proceedings of HIPS 2003, Nice, France, Apr. 2003.
- Alexandru Jugravu and Thomas Fahringer. *JavaSymphony, a Programming Model for the Grid.* in PPGaMS workshop at ICCS 2004 conference, Krakow, Poland, June 2004.
- Alexandru Jugravu and Thomas Fahringer. *Scheduling Workflow Distributed Applications in JavaSymphony.* in European Conference on Parallel Computing (Euro-Par2005), Lisboa, Portugal, Aug.-Sep. 2005.

- Alexandru Jugravu and Thomas Fahringer. *Advanced Resource Management and Scheduling of Workflow Applications in JavaSymphony* accepted for publication in the proceedings of HiPC 2005, Goa, India, Dec. 2005.

9.2 Future Work

As future work, several potential research directions are currently being considered. These are described in the following.

Security and Fault Tolerance. In developing JavaSymphony programming paradigm and middleware, security issues have not been considered. Limited fault tolerance is supported by the JavaSymphony Runtime System (Section 4.3). These are important issues for any real-world distributed system, and therefore future implementations of the JavaSymphony middleware should address them.

Grid computing. In many aspects, we can consider JavaSymphony as a programming paradigm for the Grid: it offers high-level features for resource allocation and monitoring, access to resource dynamic and static properties, code mapping and migration, and remote execution. However, JavaSymphony lacks the support for Grid security services and, in the current implementation, does not use the communication mechanisms largely accepted in the Grid community (e.g. file transfers using GridFTP [211] or SOAP-based [30] communication between Grid services).

We see two methods to tackle this issue:

- One solution is to build a new JavaSymphony middleware implementations on top of Grid services, instead of pure Java RMI. The agents of the JavaSymphony Runtime System could become Grid services and use SOAP-based messages for communication. The advantage of this solution is that it uses already existing security mechanisms for the Grid and other existing Grid services for allocating and monitoring the resources.
- A second choice would be to build an interface to existing Grid services. For example, JavaSymphony functionality for resources allocation and monitoring, or communication could make use of similar Grid services. This solution may spare programming efforts necessary for a new from-the-scratch implementation, as for the first solution.

Dynamic scheduling. Several directions can be considered for the improvement and/or diversification of our scheduling techniques. The dynamic scheduling strategy could support additional scheduling algorithms (see Section 8.2). On the other hand, the techniques can be adapted to generic Grid computing and applied to Grid workflow applications, beyond the applications built on top of the JavaSymphony programming paradigm.

Further research directions could be concerned with:

- Modelling activity execution and use prediction to estimate the activity execution times [48].

- Scheduling techniques that support pre-emption (i.e. activities can suspend execution, migrate and resume execution onto the new location).
- Investigate more generic applications (e.g. collaborative applications), which may be suitable for scheduling.

10

Appendix

10.1 Notations and Acronyms

Symbol	Description
$\mathbb{N}$	Set of natural numbers
$\mathbb{R}$	Set of real numbers
$\mathbb{R}_+$	Set of positive real numbers
$[a, b]$	Set of real numbers from a to b
$[a..b]$	Set of integer numbers from a to b
$\Longleftrightarrow$	If and only if
iff	If and only if
$\Longrightarrow$	Implication
$\rightarrow$	Function mapping
$\forall$	For all
$\exists$	Exists
$true$	True boolean value
$false$	False boolean value
$\in$	Set membership
$\notin$	Non-membership
ϕ	Empty set
$\|S\|$	Cardinality of set S
$\mathcal{P}(S)$	Power set of S
$\times$	Cross product
$\wedge$	Logical conjunction
$\vee$	Logical disjunction
$\cup$	Set union
$\cap$	Set intersection
$-$	Set difference
$\subset$	Subset of

Symbol	Description
$\overline{a}\,\langle x\rangle$, $a(x)$	Output , respectively input prefix[1]
$\alpha.P$	Prefix[1]
$P+Q$	Sum[1]
$\sum$	Sum[1]
$P\|Q$	Parallel[1]
$[x=y]P$	Match[1]
$[x\neq y]P$	Mismatch[1]
$(\nu x)P$	Restriction operator[1]
$!P$	Replication[1]
$\{P\}_m$	Process mapping[1]
c_{in}, c_{out}	Input, output channel (names)[1]
$\tilde{c}$	Set of channels (names)[1]
c_X	Channel associated to X[1]
$c_{addr,port,name}$	Channel associated to $(addr, port, name)$[1]
c_{A-B}	Channel for communication $A-B$[1]
$o\mapsto m$	Object method (process)[1]
$o\mapsto data$	Object data member (name)[1]
t, T	Workflow activity/task
m_T	Resource allocated to T
$start_T$	Start time assigned to T
$ct(t/m)$	Completion time of t, if mapped onto m
$st(t/m)$	Start time of t, if mapped onto m
$ready(t)$	Ready time of t
$avail(m)$	The availability of the machine m
$exec(t/m)$	Execution time of t, if mapped onto m
$\overline{exec(t)}$	average computation cost for t
$comm(t_1/m_1, t_2/m_2)$	communication cost for t_1 and t_2
$\overline{comm(t_1, t_2)}$	average communication cost for t_1 and t_2
Att	Set of resource attributes
att_i	Attribute function
$S(m,T,t)$	Suitability predicate
$B(T,t)$	Resource broker function
$Avail(m,t)$	Resource availability function

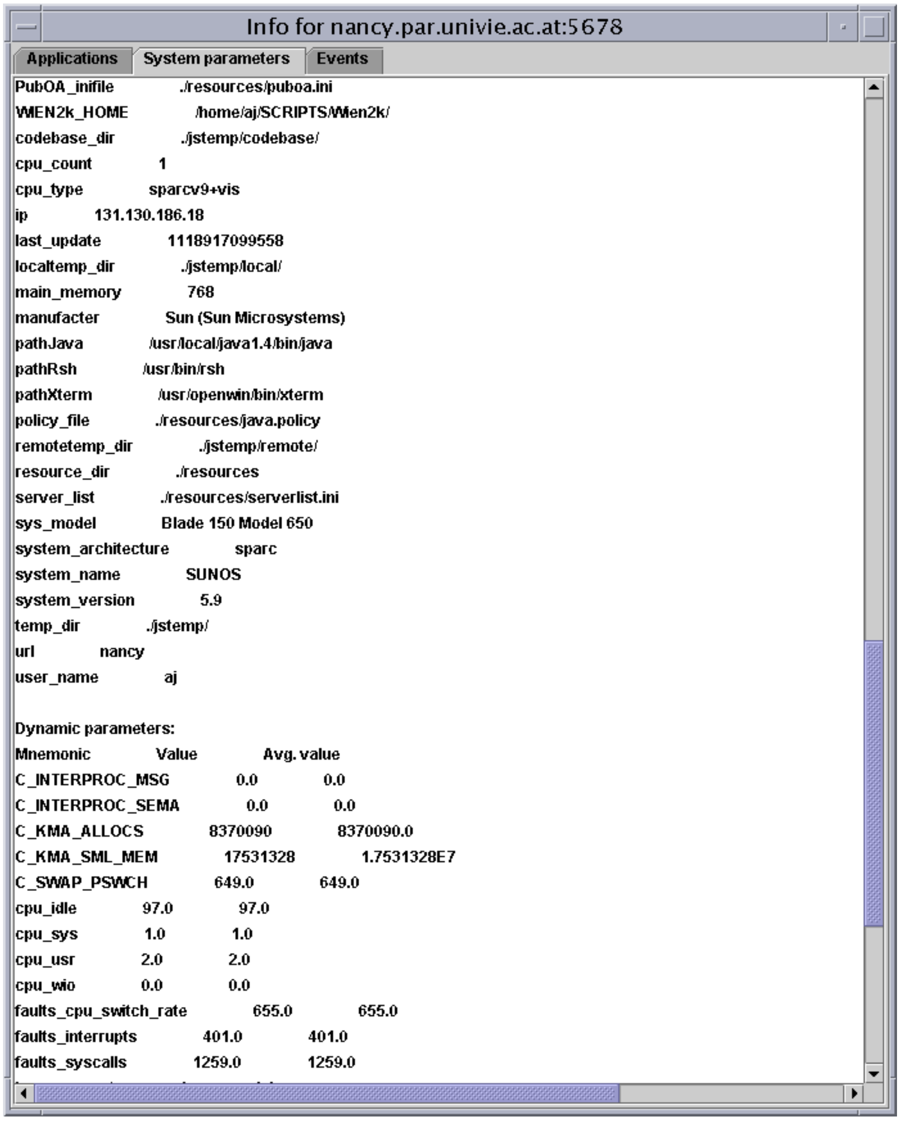
Acronym	Definition
CPU	Central Processing Unit
SISD	Single Instruction Single Data
SIMD	Single Instruction Multiple Data
MISD	Multiple Instructions Single Data
MIMD	Multiple Instruction Multiple Data
SMP	Symmetric Multi-Processor (machine, node)

[1] Pi-calculus related notations. See Appendix 10.3.

Acronym	**Definition**
MPP	Massive Parallel Processors (machine)
COW	Cluster of Workstations
NOW	Network of Workstations
LAN	Local Area Network
WAN	Wide Area Network
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SOAP	Simple Object Access Protocol
XML	eXtensible Markup Language
QoS	Quality of Service
JS	JavaSymphony
JS-Shell	JavaSymphony Administration Shell
JRS	JavaSymphony Runtime System
NA(s)	Network Agent(s)
NAS	Network Agent System
OA(s)	Object Agent(s)
OAS	Object Agent System
PubOA(s)	Public Object Agent(s)
AppOA(s)	Application Object Agent(s)
EvA(s)	Event Agent(s)
EvAS	Event Agent System
VA(s)	(Distributed) Virtual Architecture(s)
JS object(s)	JavaSymphony (distributed) objects

10.2 JavaSymphony System Parameters

Figure 10.1 shows system parameters for one computing resource displayed in a monitoring window of the JS-Shell interface.



```
PubOA_inifile            ./resources/puboa.ini
WIEN2k_HOME            /home/aj/SCRIPTS/Wien2k/
codebase_dir            ./jstemp/codebase/
cpu_count            1
cpu_type            sparcv9+vis
ip            131.130.186.18
last_update            1118917099558
localtemp_dir            ./jstemp/local/
main_memory            768
manufacter            Sun (Sun Microsystems)
pathJava            /usr/local/java1.4/bin/java
pathRsh            /usr/bin/rsh
pathXterm            /usr/openwin/bin/xterm
policy_file            ./resources/java.policy
remotetemp_dir            ./jstemp/remote/
resource_dir            ./resources
server_list            ./resources/serverlist.ini
sys_model            Blade 150 Model 650
system_architecture            sparc
system_name            SUNOS
system_version            5.9
temp_dir            ./jstemp/
url            nancy
user_name            aj

Dynamic parameters:
Mnemonic            Value            Avg. value
C_INTERPROC_MSG            0.0            0.0
C_INTERPROC_SEMA            0.0            0.0
C_KMA_ALLOCS            8370090            8370090.0
C_KMA_SML_MEM            17531328            1.7531328E7
C_SWAP_PSWCH            649.0            649.0
cpu_idle            97.0            97.0
cpu_sys            1.0            1.0
cpu_usr            2.0            2.0
cpu_wio            0.0            0.0
faults_cpu_switch_rate            655.0            655.0
faults_interrupts            401.0            401.0
faults_syscalls            1259.0            1259.0
```

Fig. 10.1. Monitoring system parameters with the JS-Shell

Table 10.3 lists the parameters supported by JavaSymphony middleware. There are three types of parameters:

- **[B]** represents parameters that are provided by benchmarks (e.g. Scimark2 benchmark [95]) that are executed at the initialization of the JavaSymphony agents onto the computing resources;
- **[S]** represents static system parameters that are determined only once, at the initialization of the JavaSymphony agents;
- **[D]** represents dynamic parameters that are updated regularly. Mean values of these parameters are computed at each update step.

The parameters values have some basic types: **[D]** - decimal (real double) value, **[I]** - integer, and **[S]** - string value. The mean values are available only for numeric values (integer or decimal); these are of decimal type always.

The availability of the parameters has been determined for two main operating systems: SunSolaris (availability value **[1]**), and Linux (availability value **[2]**). Availability value **[0]** means that the parameter does not depend on the operating system.

Table 10.3: System parameters

System Parameter name	**Type**	**Value**	**Availability**
benchCPU_composite_score	B	D	[0]
benchCPU_fft_score	B	D	[0]
benchCPU_lu_score	B	D	[0]
benchCPU_montecarlo_score	B	D	[0]
benchCPU_sor_score	B	D	[0]
benchCPU_sparsemat_score	B	D	[0]
bench_composite_score	B	D	[0]
bench_fft_score	B	D	[0]
bench_lu_score	B	D	[0]
bench_montecarlo_score	B	D	[0]
bench_sor_score	B	D	[0]
bench_sparsemat_score	B	D	[0]
Java_version	S	S	[0]
cpu_count	S	I	[0]
cpu_type	S	S	[0]
ip	S	S	[1]
last_update	S	S	[1] [2]
main_memory	S	S	[1]
manufacter	S	S	[1]
sys_model	S	S	[1]
system_architecture	S	S	[0]
system_name	S	S	[0]
system_version	S	S	[0]
url	S	S	[0]

Table 10.3: System parameters

System Parameter name	**Type**	**Value**	**Availability**
user_name	S	S	[0]
C_INTERPROC_MSG	D	D	[1]
C_INTERPROC_SEMA	D	D	[1]
C_KMA_ALLOCS	D	I	[1]
C_KMA_SML_MEM	D	I	[1]
C_SWAP_PSWCH	D	D	[1]
cpu_idle	D	D	[1] [2]
cpu_sys	D	D	[1] [2]
cpu_usr	D	D	[1] [2]
cpu_wio	D	D	[1] [2]
faults_cpu_switch_rate	D	D	[1]
faults_interrupts	D	D	[1]
faults_syscalls	D	D	[1]
jsapps_count	D	I	[0]
mem_free_kb	D	I	[1]
mem_swap_kb	D	I	[1]
procs_blocked	D	I	[1]
procs_dispatch_queue	D	I	[1]
procs_waiting	D	I	[1]
swap_allocated	D	I	[1]
swap_available	D	I	[1]
swap_reserved	D	I	[1]
swap_used	D	I	[1]
syscalls_exec	D	D	[1]
syscalls_fork	D	D	[1]
syscalls_rchar	D	D	[1]
syscalls_scall	D	D	[1]
syscalls_sread	D	D	[1]
syscalls_swrite	D	D	[1]
syscalls_wchar	D	D	[1]
unusedmem_freemem	D	I	[1]
unusedmem_freeswap	D	I	[1]
memory_allocated	D	I	[2] [1]

Table 10.4 illustrates system parameters collected from Condor machine ClassAdd. The system parameters mechanism is extensible in two ways: On the one hand, static parameters and their associated values can be added manually in JavaSymphony configuration files available for each available machine, and are simply collected by the JavaSymphony network agent at initialization. On the other hand, by building an implementation of *SystemPropertyProvider* provided by the JavaSymphony class library, new dynamic system parameters can be added to the existing ones.

Table 10.4: System parameters associated with Condor machine ClassAds

System Parameter name	**Type**	**Value**
CONDOR_Activity	S	S
CONDOR_Arch	S	S
CONDOR_COLLECTOR_HOST_STRING	S	S
CONDOR_ClockDay	S	S
CONDOR_ClockMin	S	S
CONDOR_CondorLoadAvg	S	S
CONDOR_CondorPlatform	S	S
CONDOR_CondorVersion	S	S
CONDOR_ConsoleIdle	S	S
CONDOR_CpuBusy	S	S
CONDOR_CpuBusyTime	S	S
CONDOR_CpuIsBusy	S	S
CONDOR_Cpus	S	S
CONDOR_CurrentRank	S	S
CONDOR_DaemonStartTime	S	S
CONDOR_Disk	S	S
CONDOR_EnteredCurrentActivity	S	S
CONDOR_EnteredCurrentState	S	S
CONDOR_FileSystemDomain	S	S
CONDOR_HasCheckpointing	S	S
CONDOR_HasFileTransfer	S	S
CONDOR_HasIOProxy	S	S
CONDOR_HasJICLocalConfig	S	S
CONDOR_HasJICLocalStdin	S	S
CONDOR_HasJava	S	S
CONDOR_HasMPI	S	S
CONDOR_HasPVM	S	S
CONDOR_HasRemoteSyscalls	S	S
CONDOR_JavaMFlops	S	S
CONDOR_JavaVendor	S	S
CONDOR_JavaVersion	S	S
CONDOR_KFlops	S	S
CONDOR_KeyboardIdle	S	S
CONDOR_LastBenchmark	S	S
CONDOR_LastHeardFrom	S	S
CONDOR_LoadAvg	S	S
CONDOR_Machine	S	S
CONDOR_Memory	S	S
CONDOR_Mips	S	S
CONDOR_MyAddress	S	S
CONDOR_MyType	S	S

Table 10.4: System parameters associated with Condor machine ClassAds

System Parameter name	**Type**	**Value**
CONDOR_Name	S	S
CONDOR_OpSys	S	S
CONDOR_Rank	S	S
CONDOR_Requirements	S	S
CONDOR_Start	S	S
CONDOR_StartdIpAddr	S	S
CONDOR_StarterAbilityList	S	S
CONDOR_State	S	S
CONDOR_Subnet	S	S
CONDOR_TargetType	S	S
CONDOR_TotalCondorLoadAvg	S	S
CONDOR_TotalDisk	S	S
CONDOR_TotalLoadAvg	S	S
CONDOR_TotalVirtualMachines	S	S
CONDOR_TotalVirtualMemory	S	S
CONDOR_UidDomain	S	S
CONDOR_UpdateSequenceNumber	S	S
CONDOR_UpdatesHistory	S	S
CONDOR_UpdatesLost	S	S
CONDOR_UpdatesSequenced	S	S
CONDOR_UpdatesTotal	S	S
CONDOR_VirtualMachineID	S	S
CONDOR_VirtualMemory	S	S

10.3 Pi-calculus Preliminaries

10.3.1 Basics

Prefixes	$\alpha ::=$	$\overline{a}\langle x\rangle$	output
		$a(x)$	input
		τ	silent
Processes	$P ::=$	0	Nil
		$\alpha.P$	Prefix
		$P+Q$	Sum
		$P\|Q$	Parallel
		$[x=y]P$	Match
		$[x\neq y]P$	Mismatch
		$(\nu x)P$	Restriction
		$!P$	Replication

Table 10.5. Pi-calculus syntax

The core notion of Pi-calculus[2] is the *name*. We presume that there is an infinite set of *names*, ranged over by a, b, ..., z. With *names*, we build *processes*, ranged over by P,Q ..., according to the rules in Table 10.5.

In the following, we explain the semantics of these constructs:

- The empty process 0 does not perform any action.
- The input prefix process $a(x).P$ receives the name x along the channel named a, and then continues as process P. The output prefix process $\overline{a}\langle x\rangle.P$ sends the name x along channel a and continues as process P. In some literature, the notation $\overline{a}x.P$ is used. The silent prefix process $\tau.P$ evolves into P without interactions with the environment.
- The sum process $P+Q$ acts like either P or Q. The choice between P and Q is not deterministic. Some papers present *sum* as $\sum_{i=1}^{n}\alpha_i.P_i$, which evolves into one of the P_i processes, if data is received/sent over the input/output channel α_i.
- The parallel process $P|Q$ acts as P and Q running in parallel. The processes P and Q may interact or not. The interaction means to exchange names along channels.
- The matching process $[x=y]P$ acts as P if the names x and y are the same. An alternative notation matching process is frequently used: if $(x=n)$ then P.
- Similarly, the process $[x\neq y]P$ acts as P if x and y are distinct. The Match and Mismatch operators are missing in some variants of the Pi-calculus.
- The restriction $(\nu x)P$ acts like process P in which x is *local*, meaning that no external communication can exist on channel named x.

[2] A comprehensive introduction in Pi-calculus can be found in [212]

- Replication $!P$ represents an unbound number of copies of P running in parallel: $!P \equiv P|!P \equiv P|P|\dots$

We say that the input prefix $a(x).P$ and the restriction $(\nu x)P$ *bind* the name x in P. For the output prefix $\overline{a}\langle x\rangle.P$, the name x is said to be *free* in P. We define $bn(P)$ and $fn(P)$ as the set of *bound names*, respectively *free names* in P. Note that a name could appear both bound and free in the formula for one process.

We use the notation $P[x/y]$ to denote the process resulted by replacing the name y with x.

Renaming of bound variables:		
$x(y).P \equiv x(z).(P[z/y])$ if $z \notin fv(P)$		
$(\nu x)P \equiv (\nu z)P[z/y]$ if $z \notin fv(P)$		
Abelian monoid laws for $\|$ and $+$		
$P\|Q \equiv Q\|P$	$P+Q \equiv Q+P$	commutativity
$(P\|Q)\|R \equiv P\|(Q\|R)$	$(P+Q)+R \equiv P+(Q+R)$	associativity
$P\|0 \equiv P$		
Replication		
$!P \equiv P\|!P$		
Scope extension laws		
$(\nu x)0 \equiv 0$		
$(\nu x)(P\|Q) \equiv P\|(\nu x)Q$	if $x \notin fn(P)$	
$(\nu x)(P+Q) \equiv P+(\nu x)Q$	if $x \notin fn(P)$	
$(\nu x)[u=v]P \equiv [u=v](\nu x)P$	if $x \neq u$ and $x \neq u$	
$(\nu x)[u \neq v]P \equiv [u \neq v](\nu x)P$	if $x \neq u$ and $x \neq u$	
$(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$		

Fig. 10.2. Pi-calculus structural congruence

$\overline{a}\langle y\rangle.P\|a(z).Q \to P\|Q[y/z]$		Communication
$P\|R \to Q\|R$	if $P \to Q$	Reduction under $\|$
$(\nu x)P \to Q$	if $P \to Q$	Reduction under ν
$[x=x]P \to Q$	if $P \to Q$	Match
$[x \neq y]P \to Q$	if $P \to Q$ and $x \neq y$	Mismatch
$P+Q \to R$	if $P \to R$ or $Q \to R$	Sum
$P \to Q$	if $P \equiv P' \to Q' \equiv Q$	Structural congruence

Fig. 10.3. Pi-calculus operational semantics

By using these notations, two relations over the processes can be defined. The first one is the **structural congruence** that determines which processes represent *"the same thing"* (Fig. 10.2). The second relation, called **reduction** relation, defines the operational semantics of Pi-calculus (Fig. 10.3). $P \rightarrow R$ means that P *"evolves into"* Q.

10.3.2 Variants of the Calculus

Currently, many variants of the calculus co-exist. Some of them use a subset of the constructs, which imply a simpler theory. For example the *Match* and *Mistmatch*, *Replication* are not used in some variants of the Pi-calculus. Their behaviour can be simulated by using other constructs. The sum is sometimes replaced by the guarded sum: $\sum_{i=1}^{n} \alpha_i.P_i$, which restricts the "random" choice of the process to which the sum evolves, considered unrealistic from the implementation perspective.

On the other hand, there are extensions to the classical Pi-calculus (commonly called the **monadic** Pi-calculus), which improve the expressivity of the calculus. Two of them are important for our work: polyadic Pi-calculus and higher ordered Pi-calculus [212].

The polyadic Pi-calculus

A straightforward extension is to allow the communication of arrays of names along the channels. This means to allow inputs like $a(x_1, x_2, ...x_n).P$ with $x_i \neq x_j$ if $i \neq j$, and outputs like $\overline{a} \langle y_1, y_2, ...y_n \rangle .P$.

The notation $\tilde{x}$ is used for the array of names $(x_1, x_2, ...x_n)$. The communication rule looks similar to the monadic calculus:

$$\overline{a} \langle \tilde{y} \rangle .P|a(\tilde{z}).Q \rightarrow P|Q[\tilde{y}/\tilde{z}]$$

Problems may arise if the arity of the output $\overline{a} \langle \tilde{y} \rangle$ is not equal with the arity of the input $a(\tilde{z})$. This problem is solved by using *sorts*: For each name (of a channel) a *sort* is assigned, which contains information about the type of the object (e.g. single name, array of names) that can be passed along that channel.

The polyadic input/output can be encoded with basic Pi-calculus constructs as follows:

$$a(x_1, x_2, ...x_n).Q = a(p).p(x_1).p(x_2)....p(x_n).Q \text{ with } p \notin fv(P)$$

$$\overline{a} \langle x_1, x_2, ...x_n \rangle .Q = (\nu p)\overline{a} \langle p \rangle .\overline{p} \langle x_1 \rangle .\overline{p} \langle x_2 \rangle ....\overline{p} \langle x_n \rangle ().Q \text{ with } p \notin fv(P)$$

Therefore, the theory developed for the monadic calculus is still valid for the polyadic form. The polyadic Pi-calculus also introduces *agents*, a generalization of the *process* notion. The syntax is slightly changed to include the agents, which have a more complex form than processes. We do not give more details about these changes.

The higher-order Pi-calculus

The higher-order calculus extends Pi-calculus by allowing agents/processes to be transmitted along the channels. In the higher-order Pi-calculus, the input prefix form may look like $a(X).Q$, which means that the process or agent X is received along channel a before the process evolves into Q. Q may contain X in its definition. The higher-order output prefix $\overline{a}\left\langle P\right\rangle.Q$ means that process/agent P is sent over the channel a, after which the process continues as Q.

Formally, a new category of *agent variables*, ranged over by X, Y, is introduced, with a special sort for the "agent type", and the higher-order prefix forms are added to the syntax. The *Communication* rule is adapted for the higher-order prefixes:

$$a(X).P|\overline{a}\left\langle Q\right\rangle.R \rightarrow P[Q/X]|R$$

The theory for the higher-order calculus becomes quite complex. However, formulas of the higher-order calculus could be expressed with more complex constructs of the basic calculus, and the existing theory may be applied. We can convert higher-order formulas into normal as follows: Each agent sent along a channel, is replaced by a new name, which triggers the corresponding agent. The following rules illustrate how the higher-order constructs could be eliminated from a higher-order process P. The notation $\|P\|$ stands for the formula which is obtained from P by eliminating higher-order constructs.

- $\|X\| = \overline{x}$. Each *agent variable* X is associated with a completely new name x. The process X in a formula becomes an output on channel x.
- $\|\overline{a}\left\langle Q\right\rangle.R\| = (\nu q)\overline{a}\left\langle q\right\rangle.(\|R\|\,|!q.\,\|Q\|)$. Instead of transmitting an agent Q in the higher-order output process, a new name q is sent. The process evolves into R in parallel with multiple processes $q.\,\|Q\|$ that wait for (any) input on channel p.
- $\|a(X).P\| = a(x).\,\|P\|$. A process that receives an agent X as input receives instead X an unique name x corresponding to X, and evolves to $\|P\|$.
- $\|P\|$ - each *agent variable* X is associated with a completely new name x. The occurrences of X in P in the input prefix form are replaced with outputs on channel x .

After recursively transforming the higher-order Pi-calculus formulas, the communication rule applies as follows:

$\|a(X).P\|\,|\,\|\overline{a}\left\langle Q\right\rangle.R\|$ is $a(x).\,\|P\|\,|(\nu q)\overline{a}\left\langle q\right\rangle.(\|R\|\,|!q.\,\|Q\|)$, which can be reduced to:

$$a(x).\,\|P\|\,|(\nu q)\overline{a}\left\langle q\right\rangle.(\|R\|\,|!q.\,\|Q\|) \rightarrow (\nu q)(\|P\|\,[q/x]|\,\|R\|\,|!q.\,\|Q\|)$$

P is supposed to contain occurrences of X, which are replaced by outputs $\overline{x}$ and then by $\overline{q}$. Each such occurrence is reduced by communication with one

$q.\,\|Q\|$ process, such that we finally get $\|P[Q/X]\|$ instead of P. Therefore, we may reduce the entire formula to:

$$(\nu q)\,\|P[Q/X]\|\,|\,\|R\|\,|!q.\,\|Q\|$$

and P and R do not contain q. Consequently, the replicated input $!q.\,\|Q\|$ does not interact anymore. The conclusion is that the two formulas, the higher-order formula and its transformation, express the same process behaviour.

A variant of the Pi-calculus for JavaSymphony

The Pi-calculus variants discussed above are still at a very low level. We propose extensions to Pi-calculus in order to express the functionality of JavaSymphony at a higher level, closer to a programming language. At the same time, we want to preserve the properties of the classical calculus, in order to be able to apply the existing Pi-calculus theory.

Objects in Pi-calculus. The object oriented programming paradigm is considered to be at an acceptable high level. Expressing object instances in a formal language is an important advantage.

An object can be seen as the combination of data members and methods. In terms of Pi-calculus, this can be translated in *names* for object's data members and processes for the object's methods. We consider *obj* to range over the set of object instances. A specific sort is associated to each object type. Our intention is to use output and input prefix forms such as $a(obj).P$ and $\overline{a}\,\langle obj\rangle\,.Q$. On the one hand, the polyadic form of Pi-calculus allows us to send multiple names along one channel at the same time, which is similar to sending complex data structures. On the other hand, the higher-order Pi-calculus enables the sending of processes along the channels. We consider an object instance represented as an array of member data and member methods: $(x_1, x_2, ...x_n, M_1, M_2, ...M_m)$, with x_i data member is represented as a *name* and M_j method of the object class is represented as a process. The combination of polyadic Pi-calculus and higher-order Pi-calculus allows such a construct to be send/received along the channels.

The notation $obj \mapsto x_i$ denotes the x_i data member of the object *obj*, which is a *name*. We denote by $obj \mapsto M_j$ (or $Execute(obj \mapsto M_j)$) the process associated with the execution of the method M_j of *obj*. At this point, we ignore the method parameters. The following short example demonstrates the use of the new constructs:

$$a(obj).Execute(obj \mapsto M1).\overline{b}\,\langle obj \mapsto id\rangle\,|\overline{a}\,\langle obj1\rangle\,.b(id)$$

For this example, we have two processes running in parallel. The first one receives an object *obj* (an object variable) along channel a, executes one of its methods $M1$ and then sends *id* object member along channel b. The second process sends *obj*1 (an object instance) along channel a and then waits to receive a name *id* on channel b. This extension does not affect the semantics of the calculus, since the new construct can be expressed in terms of polyadic or higher-order Pi-calculus.

Mapping processes onto machines. We introduce new constructs to express the mapping of a computing process (represented by a Pi-calculus formula) onto a computing resource. For a finite set of computing resources $M = \{m_1, m_2, ...m_n\}$, we use the notation $\{P\}_{m_i}$ to express the fact that *"the process P takes place on machine m_i"* . We can extend the notation to express the fact that *"several processes are running in parallel on distinct machines"*. We call *mapping* the association of a process/sub-process to a resource.

This extension represents an important gain in expressivity of the calculus. However, a new set of semantic rules is required. The new rules should not interfere with the Pi-calculus existing rules.

For example, three processes running in parallel as in:

$$a(x).P|a\langle y\rangle.b(z)Q|b\langle w\rangle.R$$

could be mapped onto distinct machines:

$$\{a(x).P\}_{m1}|\{a\langle y\rangle.b(z)Q\}_{m2}|\{b\langle w\rangle.R\}_{m3}$$

or onto a single machine:

$$\{a(x).P|a\langle y\rangle.b(z)Q|b\langle w\rangle.R\}_{m1}$$

In terms of Pi-calculus, the formulas represent the same process. However, the two formulas differ in terms of resource usage. The Pi-calculus rules for structural congruence and reduction relation are still valid, because they are not affected by the mapping of the processes. This means that: $P \equiv Q$ implies $\{P\}_m \equiv \{Q\}_m$ and $P \rightarrow Q$ implies $\{P\}_m \rightarrow \{Q\}_m$. In fact, we can omit the mapping in reasoning about the process behaviour, when this is not relevant.

However, we can assume that the reduction preserves the mapping, which means that the processes cannot migrate from a resource to another. This property is expressed as the following formal rule:

if $P|Q \rightarrow R$ then exist P_1 and Q_1 such as $P \rightarrow P_1$, $Q \rightarrow Q_1$, $R \equiv P_1|Q_1$ and $\{P\}_m\,|\,\{Q\}_n \rightarrow \{P_1\}_m\,|\,\{Q_1\}_n$

Postfixes and sequence of processes. To the best of our knowledge, Pi-calculus (or any of its variants) does not use a construct to express a *sequence of processes*. Basically, we intend to use a construct $R = P.Q$, meaning that R behaves like P until P is reduced to 0 and thereafter it starts to behave like Q. We call R a *sequence process*.

Formally written, this means that:

if $P \rightarrow 0$, $R = P.Q$ then $R \rightarrow Q$

We analyse a few examples to motivate this construct and point our several potential problems with this extension to Pi-calculus.

- If P is a sequence of prefixes:
  $P = \alpha_1.\alpha_2...\alpha_n$ where α_i is $a_i(x)$, $\overline{a_i}\langle x\rangle$ or τ.
  then $P.Q$ is allowed in the standard Pi-calculus, as well.

- Suppose $P = A_1 + A_2$ (sum) or $P = A_1|A_2$ (parallel construct). In this case, there is no equivalent in the standard Pi-calculus for P.Q
- Supposing that $P =!A$ (replication). In this case, there is no equivalent in the standard Pi-calculus for P.Q. Since P could never be reduced to 0, we may say that P.Q will always behave like P.
- $P = a(x)|\overline{b}\langle y\rangle$ and $R = \overline{a}\langle z\rangle .b(x).R'$ note that P could reduce to 0 in parallel with R. Therefore, we may apply the reduction to $P.Q|R$:

  $$P.Q|R \to Q[z/x]|R'[y/x]$$

  Note that $a(x).b\langle y\rangle .Q|R$ simulates the behaviour of $P.Q|R$, but $b\langle y\rangle .a(x).Q|R$ does not.

On the other hand, we could use a simpler form of the sequence construct, by allowing the *postfix process*, written as $P.\alpha$, where α may be $a(x)$, $\overline{a}\langle x\rangle$ or τ, similar to prefix processes defined in Table 10.5.

Allowing sequence constructs implies allowing *postfix processes*. This becomes clear, by choosing $Q = \alpha$ in $P.Q$. On the other hand, the sequence construct $P.Q$ may be simulated by using the postfix construct: $(\nu q)P.\overline{q}\langle\rangle |q(x).Q$. After P finishes, data is sent along a private channel q to activate Q.

Therefore, we conclude that using sequence processes is equivalent to using postfix processes.

10.4 XML Schema for JavaSymphony Workflow Specification Language

Figures[3] 10.4, 10.6, 10.5, 10.7, 10.8, 10.9 show the XML Schema that defines the elements of the JavaSymphony workflow specification language.

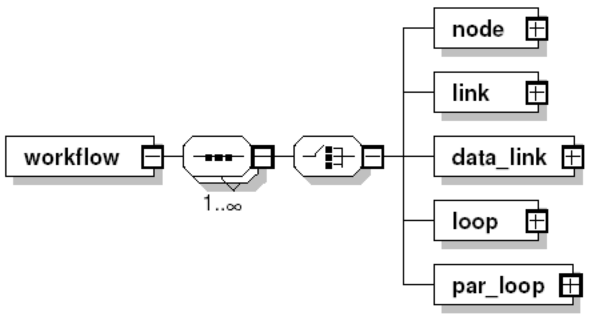


Fig. 10.4. XML Schema for workflow definition

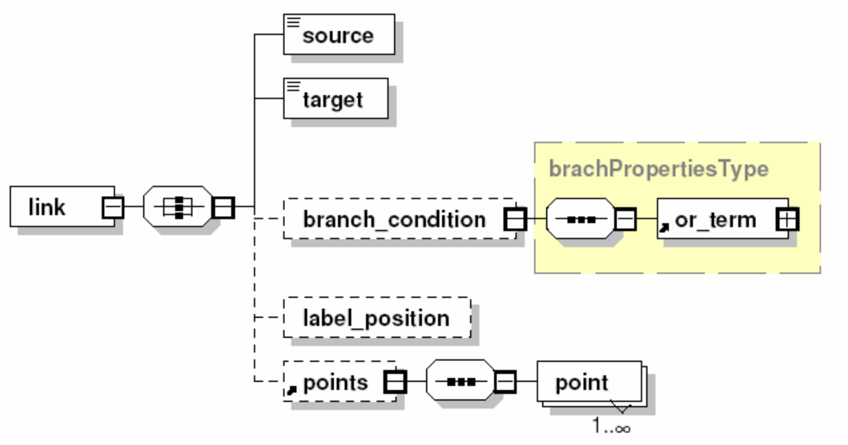


Fig. 10.5. XML Schema for workflow definition. Link element

[3] The figures have been generated using Altnova XMLSpy Home Edition (www.altnova.com)

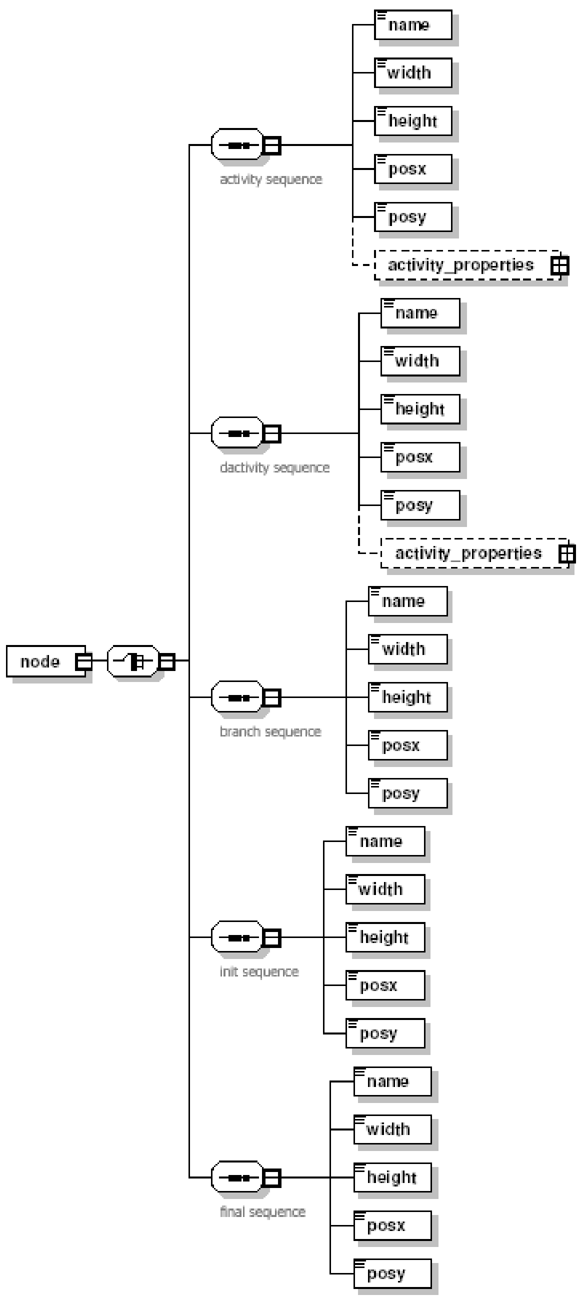


Fig. 10.6. XML Schema for workflow definition. Node element

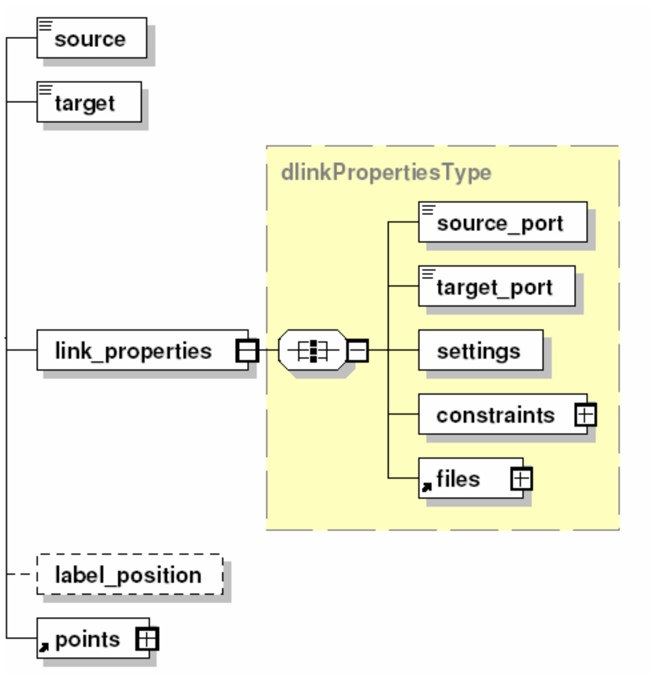


Fig. 10.7. XML Schema for workflow definition. Data-link element

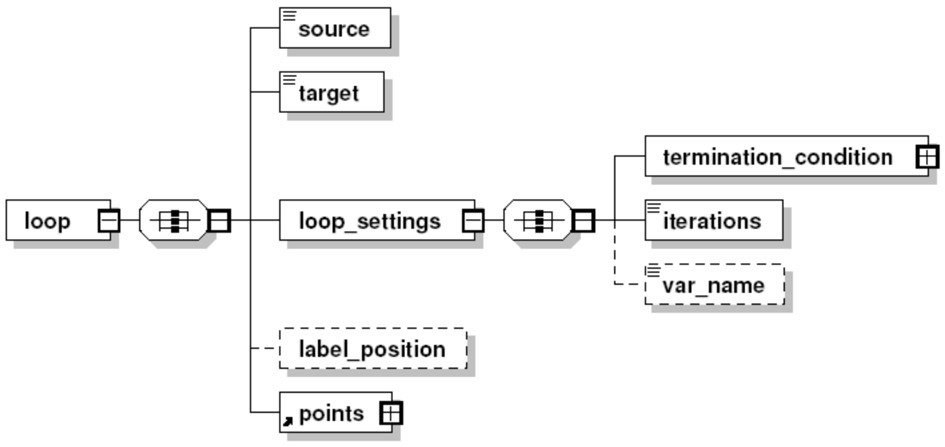


Fig. 10.8. XML Schema for workflow definition. Loop element

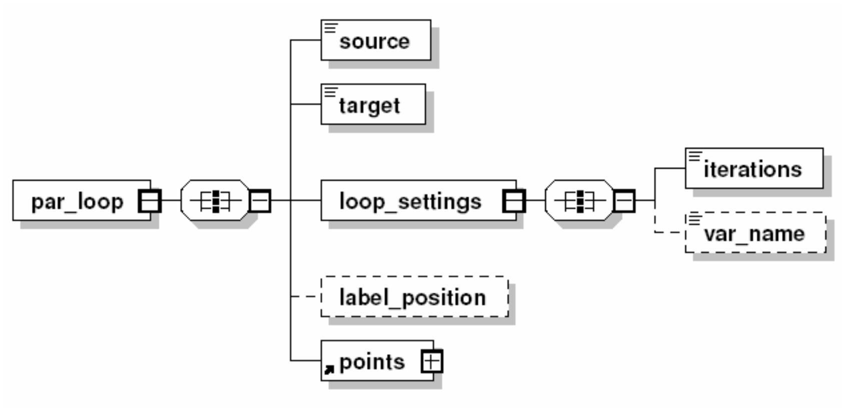


Fig. 10.9. XML Schema for workflow definition. Link element

References

1. Designing and Building Parallel Programs. *Designing and Building Parallel Programs*. Addison-Wesley, 1995. available online at:http://www-unix.mcs.anl.gov/dbpp/.
2. Bart Jacob, Luis Ferreira, Norbert Bieberstein, Candice Gilzean, Jean-Yves Girard, Roman Strachowski, and Seong (Steve) Yu. Enabling Applications for Grid Computing with Globus. IBM Red Book, available at www.ibm.com/redbooks.
3. A Brief History of the Internet. available at: http://www.isoc.org/internet/history/brief.shtml. last acces date: Jul 25 2005.
4. A Little History of the World Wide Web. available at: http://www.w3.org/History.html. last acces date: Jul 25 2005.
5. Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *The Java Language Specification, Second Edition*. Sun Microsystems, 2000.
6. Java technology. http://java.sun.com/. last acces date: Aug 25 2005.
7. J.M. Bull, L.A. Smith, M.D. Westhead, D.S. Henty, and R.A. Davey. A benchmark suite for high performance java. *Concurrency: Practice and Experience*, 2000.
8. Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan-Kaufman, San Francisco, CA, USA, 1st edition, 1998.
9. Madhu Chetty and Rajkumar Buyya. Weaving computational Grids: How analogous are they with electrical grids? *Computing in Science and Engineering*, 4(4):61–71, Jul - Aug 2002.
10. Geoffrey Fox Fran Berman and Tony Hey, editors. *Grid Computing: Making the Global Infrastructure a Reality*. Wiley, March 2003.
11. I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration, 2002.
12. I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.
13. The globus alliance. http://www.globus.org/. last acces date: Aug 25 2005.
14. Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, January/March 1998.

15. Message Passing Interface Forum. *Document for a Standard Message Passing Interface*, draft edition, November 1993.
16. C. Sivula. A call for distributed computing (RPC). *Datamation*, 36(1):75–76, 78, 80, January 1990.
17. Ward Rosenberry and Jim Teague. *Distributing Applications Across DCE and Windows NT*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, December 1993.
18. Nat Brown and Charlie Kindel. Distributed Component Object Model Protocol - dcom/1.0. Microsoft Corporation Internet-Draft, URL:http://www.microsoft.com/oledev/olecom/draft-brown-dcom-v1-spec-02.txt, January 1998.
19. David S. Linthicum. CORBA 2.0? *Open Computing*, 12(2):68–??, Feb 1995.
20. William Grosso. *Java RMI: Designing and building distributed applications.* O'Reilly & Associates, Inc., Cambridge, MA 02140, USA, 2002.
21. Oscar H. Ibarra and Chul E. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *J. ACM*, 24(2):280–289, 1977.
22. WfMC. Workflow Management Coalition: http://www.wfmc.org/ , 2003.
23. Michael J. Flynn. Some Computer Organizations and Their Effectiveness. In *IEEE Trans. Computers*, volume C-21, pages 948–960. Sept. 1972.
24. V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* Benjamin/Cummings, Redwood City, 1994.
25. Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms.* Prentice Hall, Jan. 2002.
26. Aad J. van der Steen and Jack J. Dongarra. Overview of recent supercomputers. available at: http://www.phys.uu.nl/ euroben/reports/web05a/overview.html. last acces date: June 15 2005.
27. Gregory F. Pfister. *In Search of Clusters: The Ongoing Battle in Lowly Parallel Computing.* Prentice Hall, 1998.
28. Karl Czajkowski, Ian Foster, Nick Karonis, Carl Kesselman, Stewart Martin, Warren Smith, and Steve Tuecke. A Resource Management System for Metacomputing Systems. In *PPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82, 1998.
29. Timothy. G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *A Pattern Language for Parallel Programming.* Addison Wesley Software Patterns Series, 2004.
30. Arthur Ryman. Simple object access protocol (soap) and web services. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, page 689, Washington, DC, USA, 2001. IEEE Computer Society.
31. Wikipedia. The Free Encyclopedia. available at: http://en.wikipedia.org/. last acces date: June 15 2005.
32. Workflow Management Coalition Terminology & Glossary. The Workflow Management Coalition Specification, Doc. WFMC-TC-1011, Feb. 1999.
33. David Kra. Six strategies for grid application enablement. available at http://www-128.ibm.com/developerworks/grid/library/gr-enable/, Apr 2004.
34. R. Milner. Communicating and mobile systems: The pi-calculus. University Press, Cambridge, UK, 1999.
35. Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Siebel Systems,

Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business process execution language for web services (bpel4ws). Specification version 1.1, Microsoft, BEA, and IBM, May 2003.

36. Dietmar W. Erwin and David F. Snelling. UNICORE: A Grid computing environment. *Lecture Notes in Computer Science*, 2150:825–??, 2001.
37. The Condor Team. Dagman (directed acyclic graph manager). http://www.cs.wisc.edu/condor/dagman/.
38. Sriram Krishnan, Patrick Wagstrom, and Gregor von Laszewski. GSFL : A Workflow Framework for Grid Services. Technical Report, Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL 60439, U.S.A., July 2002.
39. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(3):5–51, July 2003.
40. Howard Smith and Peter Fingar. Workflow is just a Pi. http://www.bpm3.com/picalculus), Nov. 2003.
41. M. Dumas and A. Hofstede. UML Activity Diagrams as a Workflow Specification Language. In *4th International Conference on UML, LNCS 2185*, Toronto, Canada, October 2001. Springer Verlag.
42. Tracy D. Braun, Howard Jay Siegel, Noah Beck, Ladislau Bni, Muthucumaru Maheswaran, Albert I. Reuther, James P. Robertson, Mitchell D. Theys, Bin Yao, Debra A. Hensgen, and Richard F. Freund. A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems. In *Heterogeneous Computing Workshop*, pages 15–29, 1999.
43. Ken Kennedy et.al. New Grid Scheduling and Rescheduling Methods in the GrADS Project. In *International Parallel and Distributed Processing Symposium, Workshop for Next Generation Software*, Santa Fe, New Mexico, April 2004. IEEE Computer Society Press.
44. Muthucumaru Maheswaran, Shoukat Ali, Howard Jay Siegel, Debra Hensgen, and Richard F. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *Proceedings of the Eighth Heterogeneous Computing Workshop*, page 30. IEEE Computer Society, 1999.
45. B. Kruatrachue and T.G. Lewis. Duplication scheduling heuristics (dsh): A new precedence task scheduler for parallel processor systems. Technical Report OR 97331, Oregon State University, Corvallis, 1987.
46. Yu-Kwong Kwok and Ishfaq Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59(3):381–422, 1999.
47. H. Topcuoglu, S. Hariri, and M.-Y. Wu. Task scheduling algorithms for heterogeneous processors. In *Eighth Heterogeneous Computing Workshop*, pages 3–14. IEEE C.S. Press, 1999.
48. T. Fahringer. *Automatic Performance Prediction of Parallel Programs.* Kluwer Academic Publishers, Boston, USA, ISBN 0-7923-9708-8, March 1996.
49. Andrei Radulescu and Arjan J.C. van Gemund. Low-cost task scheduling for distributed-memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 13(6), June 2002.
50. Y.C. Chung and S. Ranka. Applications and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed memory multiprocessors. In *Supercomputer '92*, 1992.

51. I. Ahmad and Y.-K. Kwok. On exploiting task duplication in parallel program scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 9(9):872–892, 1998.
52. Christos Papadimitriou and Mihalis Yannakakis. Towards an architecture-independent analysis of parallel algorithms. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 510–513. ACM Press, 1988.
53. J.Y. Colin and P. Chretienne. C.p.m. scheduling with small computation delays and task duplication. In *Operations Research*, pages 680–684, 1991.
54. H. Chen, B. Shirazi, and J. Marquis. Performance evaluation of a novel scheduling method: Linear clustering with task duplication. In *Proceedings of International Conference on Parallel and Distributed Systems*, Dec. 1993.
55. 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, 1996.
56. Gyung-Leen Park, Behrooz Shirazi, and Jeff Marquis. Dfrn: A new approach for duplication based scheduling for distributed memory multiprocessor systems. In *Proceedings of the 11th International Symposium on Parallel Processing*, pages 157–166. IEEE Computer Society, 1997.
57. Min-You Wu and Daniel Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):330–343, 1990.
58. G.-L. Parkand B. Shirazi, J. Marquis, and H. Choo. Decisive path scheduling: A new list scheduling method. In *Proc. of the Int. Conf. on Parallel Processing*, pages 472–480, 1997.
59. Thomas L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Commun. ACM*, 17(12):685–690, 1974.
60. Yu-Kwong Kwok, Ishfaq Ahmad, and Jun Gu. Fast : A low-complexity algorithm for efficient scheduling of dags on parallel processors. In *25th International Conference on Parallel Processing*, volume 2, pages 150–157, Aug. 1996.
61. Behrooz Shirazi, Mingfang Wang, and Girish Pathak. Analysis and evaluation of heuristic methods for static task scheduling. *J. Parallel Distrib. Comput.*, 10(3):222–2232, 1990.
62. J.J. Hwang, Y.C. Chow, F.D. Anger, and C.Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *Journal on Computing*, 18(2), 1989.
63. Chung Yee Lee, Jing Jang Hwang, Yuan Chieh Chow, and Frank D. Ange. Multiprocessor scheduling with interprocessor communication delays. *Operations Research Letters*, 7:141–147, June 1988.
64. 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.
65. A. Dogan and F.Ozguner. LDBS: A duplication based scheduling algorithm for heterogeneous computing systems. In *Int'l Parallel Processing (ICCP'02)*, 2002.
66. G. C. Sih and E. A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Trans. Parallel Distrib. Syst.*, 4(2):175–187, 1993.

67. Hyunok Oh and Soonhoi Ha. A static scheduling heuristic for heterogeneous processors. In *Euro-Par, Vol. II*, pages 573–577, 1996.
68. M. Maheswaran and H. Siegel. A dynamic matching and scheduling algorithm for heterogeneous computing systems, 1998.
69. Hesham El-Rewini and T. G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *J. Parallel Distrib. Comput.*, 9(2):138–153, 1990.
70. M. Iverson, F. Ozguner, and G. Follen. Parallelizing existing applications in a distributed heterogeneous environment, 1995.
71. S. Ranaweera and D.P. Agrawal. A task duplication based scheduling algorithm for heterogeneous systems. In *14 th International Parallel and Distributed Processing Symposium (IPDPS'00),Cancun,Mexico*, May 2000.
72. Yu-Kwong Kwok. Parallel program execution on a heterogeneous pc cluster using task duplication. In *9th Heterogeneous Computing Workshop,Cancun,Mexico*, pages 364–374, May 2000.
73. Andrei Radulescu and Arjan J. C. van Gemund. Fast and effective task scheduling in heterogeneous systems. In *Heterogeneous Computing Workshop*, pages 229–238, 2000.
74. Sanjeev Baskiyar and Prashanth C. SaiRanga. Scheduling directed a-cyclic task graphs on heterogeneous network of workstations to minimize schedule length. In *Proc. of International Conference on Parallel Processing Workshops,Kaohsiung, Taiwan*, Oct. 2003.
75. Radu Prodan. *Experiment Management, Performance Optimisation, and Tool Integration in Grid Computing.* PhD thesis, Wien Fakultat fur Informatik, 2004.
76. G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Conference*, pages 483–485, 1967.
77. J. Sherwani, N. Ali, N. Lotia, Z. Hayat, and R. Buyya. Libra: An economy driven job scheduling system for clusters, 2002.
78. Tan Tien Ping, Gian Chand Sodhy, Chan Huah Yong, and Fazilah Haron andRajkumar Buyya. A Market-Based Scheduler for JXTA-Based Peer-to-Peer Computing System. *Lecture Notes in Computer Science*, 3046:147–157, Apr 2004.
79. Gilbert Strang. The discrete cosine transform. *SIAM Review*, 41(1):135–147, March 1999.
80. Gescher homepage: http://gescher.vcpc.univie.ac.at/.
81. W.T. Vetterling, S.A. Teukolsky, W.H. Press, and B.P. Flannery. *Numerical Recipes: Example Book (FORTRAN).* Cambridge University Press, 1990.
82. Donald E. Knuth. *The Art of Computer Programming.Fundamental Algorithms*, volume 1. Addison-Wesley, 1997.
83. R. Aversa, B. Di Martino, N. Mazzocca, and S. Venticinque. Mobile agents for distribute and dynamically balanced optimization applications. in High-Performance Computing and Networking(Lecture Notes in Computer Science vol.2119), ed. By B. Hertzberger et al. (Springer, Berlin, 2001) pp.161-170, 2001.
84. G. Stefanescu. "Interactive systems": - from folklore to mathematics. In *Proc. 6th International Workshop on Relational Methods in Computer Science*, pages 208–221, Oisterwijk (near Tilburg),The Netherlands, 2001. Also Springer LNCS 2002, to appear.

85. Michael J. Wiener. Efficient DES key search, technical report TR-244, Carleton University. In *William Stallings, Practical Cryptography for Data Internetworks*. IEEE Computer Society Press, 1996.
86. Javaparty homepage: http://www.ipd.uka.de/javaparty/. last acces date: Sep 17 2003.
87. Proactive homepage: http://www-sop.inria.fr/sloop/javall/. last acces date: Sep 17 2003.
88. R. Aversa, B. Di Martino, N. Mazzocca, M. Rak, and S. Venticinque. Integration of mobile agents and openmp for programming clusters of shared memory processors: a case study. accepted for publication in proc. of PaCT 2001 Conference, 8-12 Sept. 2001, Barcelona, Spain, 2001.
89. E. Laure and H. Moritsch. A High Performance Decomposition Solver for Portfolio Management Problems in the AURORA Financial Management System. Technical Report TR01-09, Institute for Software Science, University of Vienna, October 2001.
90. G. C. Pflug and A. Swietanowski. Selected parallel optimization methods for financial management under uncertainty. In *Parallel Comput., 26:3-25*. 2000.
91. A. Ruszczynski. Decomposition methods in stochastic programming. In *Math. Programming, 79:333-353*. 1997.
92. A. Ruszczynski. Parallel decomposition of multistage stochastic programming problems. In *Math. Programming, 58: 201-228*. 1993.
93. E. Laure and H. Moritsch. Portable Parallel Portfolio Optimization in the Aurora Financial Management System. In *Proceedings of SPIE ITCom 2001 Conference: Commercial Applications for High-Performance Computing*, Denver, Colorado, August 2001.
94. H.Moritsch and G.Ch.Pflug. Java Implementation of Asynchronous Parallel Nested Optimization Algorithms. In *Proceedings of the 3rd Workshop on Java for High Performance Computing, Sorrento, Italy*, June 2001.
95. Scimark2 benchmark. available at: http://math.nist.gov/scimark2. last acces date: June 10 2004.
96. P.Blaha, K.Schwarz, G.Madsen, D.Kvasnicka, and J.Luitz. *WIEN2k: An Augmented Plane Wave plus Local Orbitals Program for Calculating Crystal Properties*. Vienna University of Technology, 2001.
97. Wasim homepage: http://www.nccr-climate.unibe.ch/download/wp3 /p32/p32_wasim.html. last acces date: Mar 1 2005.
98. Invmod homepage: http://dps.uibk.ac.at/ marek/projects/invmod_wasim/. last acces date: Mar 1 2005.
99. Montage homepage: http://montage.ipac.caltech.edu. last acces date: Mar 1 2005.
100. Hong-Linh Truong and Thomas Fahringer. Online performance monitoring and analysis of grid scientific workflows. In *European Grid Conference 2005 (EGC2005),Amsterdam, The Netherlands*. LNCS, Springer-Verlag, February 14 -16 2005.
101. V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: practice and experience*, 2(4):315–339, December 1990.
102. Andrew S. Grimshaw, William A. Wulf, James C. French, Alfred C. Weaver, and Paul F. Reynolds, Jr. Legion: The next logical step toward a nationwide virtual computer. Technical Report CS-94-21, Department of Computer Science, University of Virginia, June 08 1994. Mon, 28 Aug 1995 21:06:39 GMT.

103. Henri Casanova and Jack Dongarra. NetSolve: A network-enabled server for solving computational science problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, Fall 1997.
104. Ken Arnold, Ann Wollrath, Bryan O'Sullivan, Robert Scheifler, and Jim Waldo. *The Jini specification*. Addison-Wesley, Reading, MA, USA, 1999.
105. Ann Wollrath, Jim Waldo, and Roger Riggs. Java-centric distributed computing: Providing a homogeneous view of a heterogeneous group of machines. *IEEE Micro*, 17(3), May/June 1997.
106. Rob van Nieuwpoort, Jason Maassen, Heri E. Bal, Thilo Kielmann, and Ronald Veldema. Wide-area parallel computing in Java. In *Proceedings of the ACM Java Grande Conference*, New York, NY, June 1999. ACM Press.
107. Y. Aridor, M. Factor, and A. Teperman. cJVM: A single system image of a JVM on a cluster. In *International Conference on Parallel Processing ICPP*, pages 4–11, Aizu-Wakamatsu,Fukushima, Japan, Sep. 21-24 1999. Springer Verlag,Heidelberg Germany.
108. M. Philippsen and B. Haumacher. More Efficient Object Serialization. *Lecture Notes in Computer Science*, 1586, 1999.
109. C. Nester, M. Philippsen, and B. Haumacher. A more efficient RMI. In *Proceedings of the ACM Java Grande Conference, San Francisco, CA.*, pages 152–159, New York, NY, June 1999. ACM Press.
110. Eli Tilevich and Yannis Smaragdakis. NRMI: Natural and efficient middleware. In *International Conference on Distributed Computer Systems (ICDCS)*, pages 252–261, 2003.
111. Fabian Breg, Shridhar Diwan, Juan Villacis, Jayashree Balasubramanian, Esra Akman, and Dennis Gannon. Java RMI performance and object model interoperability: experiments with Java/HPC++. *Concurrency: Practice and Experience*, 10(11–13):941–955, 1998.
112. X. Chen and V. H. Allan. MultiJav: A distributed shared memory system based on multiple java virtual machines. In *Proc. of the Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, volume I, pages 91–98, Las Vegas, Nevada, USA, July 13 - 16 1998. CSREA Press (ISBN).
113. Ronald Veldema, Rutger F. H. Hofman, Raoul Bhoedjang, Ceriel J. H. Jacobs, and Henri E. Bal. Source-level global optimizations for fine-grain distributed shared memory systems. In *Eight ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP)*, Snowbird, Utah, June 18-20 2001. ACM Press, New York, NY, USA.
114. Weimin Yu and Alan Cox. Java/DSM: A platform for heterogeneous computing. *Concurrency: Practice and Experience*, 9(11):1213–1224, November 1997.
115. Matchy J. M. Ma, Cho-Li Wang, and Francis C. M. Lau. JESSICA: Java-Enabled Single-System-Image Computing Architecture. *Parallel and Distributed Computing*, 60(10):1194–1222, Oct. 2000.
116. Wenzhang Zhu, Weijian Fang, Cho-Li Wang, and Francis C.M. Lau. A New Transparent Java Thread Migration System Using Just-in-Time Recompilation. In *The 16th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2004)*, pages 766–771, MIT Cambridge, MA, USA, Nov. 9-11 2004.
117. Jikes Research Virtual Machine (RVM). http://jikesrvm.sourceforge.net/. last acces date: Aug 21 2005.

118. Han-Ku Lee Bryan Carpenter, Geoffrey Fox and Sang Boem Lim. Applications of HPJava. In *16th International Workshop on Languages and Compilers for Parallel Computing*, volume LNCS 2958. Springer Verlag, Oct. 2003.
119. Kees van Reeuwijk, Arjan J. C. van Gemund, and Henk J. Sips. Spar: A programming language for semi-automatic compilation of parallel programs. *Concurrency: Practice and Experienc*, 9(11):1193–1205, 1997.
120. Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: High-performance java dialect. *Concurrency: Practice and Experience*, 10(11-13):825–836, 1998.
121. JavaPVM home page. http://www.isye.gatech.edu/chmsr/JavaPVM. last acces date: Aug 21 2005.
122. Adam J. Ferrari. JPVM: Network Parallel Computing in Java. In *ACM 1998 Workshop on Java for High-Performance Network Computing, Palo Alto*, Feb. 1998.
123. B.Y. Zhang, G.W. Yang, and W.M. Zheng. Jcluster: An Efficient Java Parallel Environment in a Large-scale Heterogeneous Cluster. *Concurrency and Computation: Practice and Experience*, 2000.
124. Mark Baker, Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Sang Lim. mpiJava: An Object-Oriented Java interface to MPI. In *International Workshop on Java for Parallel and Distributed Computing, IPPS/SPDP 1999*, San Juan, Puerto Rico, April 1999.
125. Luis F. G. Sarmenta, Satoshi Hirano, and Stephen A. Ward. Towards Bayanihan: Building an extensible framework for volunteer computing using Java. In ACM, editor, *ACM 1998 Workshop on Java for High-Performance Network Computing*, New York, NY, USA, 1998. ACM Press.
126. A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff. Charlotte: metacomputing on the Web. In K. Yetongnon and S. Hariri, editors, *Proceedings of the ISCA International Conference. Parallel and Distributed Computing Systems, Dijon, France, 25–27 September, 1996*, volume 1, Raleigh, NC, USA, 1996. International Society of Computers and Their Applications (ISCA).
127. N. Nisan, S. London, O. Regev, and N. Camiel. Globally distributed computation over the internet-the POPCORN project, 1998.
128. Michael O. Neary and Peter Cappello. Advanced Eager Scheduling for Java-Based Adaptively Parallel Computing. *Concurrency and Computation: Practice and Experience*, 17:797–819, 2005.
129. Hiromitsu Takagi, Satoshi Matsuoka, Hidemoto Nakada, Satoshi Sekiguchi, Mitsuhisa Satoh, and Umpei Nagashima. Ninflet: a migratable parallel objects framework using Java. In ACM, editor, *ACM 1998 Workshop on Java for High-Performance Network Computing*, New York, NY 10036, USA, 1998. ACM Press.
130. Maurice Herlihy and Michael P. Warres. A tale of two directories: implementing distributed shared objects in java. *Concurrency - Practice and Experience*, 12(7):555–572, 2000.
131. Parabon Computation - Computing outside the box. http://www.parabon.com/. last acces date: Aug 21 2005.
132. Peter Cappello and Dimitrios Mourloukos. CX: A Scalable, Robust Network for Parallel Computing. *Scientific Programming*, 10(2):159 – 171, 2001. Ewa Deelman and Carl Kesselman eds.

133. A. Alexandrov, M. Ibel, K. Schauser, and C. Scheiman. SuperWeb: Towards a global web-based parallel computing infrastructure. In *The 11th IEEE International Parallel Processing Symposium (IPPS)*, pages 100–106, Geneva, Switzerland, April 1-5 1997. IEEE Computer Society.
134. S. Lalis and A. Karipidis. JaWS: An open market-based framework for distributed computing over the internet. In *IEEE/ACM International Workshop on Grid Computing (GRID 2000)*. Springer Verlag, Dec. 2000.
135. Paolo Ciancarini and Davide Rossi. Jada - Coordination and Communication for Java Agents. In Jan Vitek and Christian Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 213–228. Springer-Verlag: Heidelberg, Germany, April 1997.
136. Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces principles, patterns, and practice*. Addison-Wesley, Reading, MA, USA, 1999.
137. Danny B. Lange and Mitsuru Oshima. *Programming and Deploying Mobile Agents with Java Aglets*. Addison-Wesley, Reading, MA, USA, September 1998.
138. Voyager. http://www.recursionsw.com/voyager.htm. last acces date: Aug 21 2005.
139. Michael Philippsen and Matthias Zenger. JavaParty — transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997.
140. Denis Caromel, Wilfried Klauser, and Julien Vayssière. Towards seamless computing and metacomputing in Java. *Concurrency: Practice and Experience*, 10(11–13):1043–1061, 1998.
141. M. Izatt, P. Chan, and T. Brecht. Ajents: Towards an environment for parallel, distributed and mobile java applications. In *Proceedings of ACM 1999 Java Grande Conferernce*, pages 15–25, San Francisco, CA, June 1999.
142. Michael Philippsen, Bernhard Haumacher, and Christian Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, May 2000.
143. Mohammed M. Fuad and Michael J. Oudshoorn. Adjava - automatic distribution of java applications. In Michael J. Oudshoorn, editor, *25th Australasian Computer Science Conference (ACSC2002)*, volume 4 of *Conferences in Research and Practice in Information Technology*, pages 65–75, Melbourne, Australia, 2002. ACS.
144. Tao Yang and Apostolos Gerasoulis. List scheduling with and without communication delays. *Parallel Computing*, 19(12):1321–1344, 1993.
145. Vivek Sarkar. Partitioning and scheduling parallel programs for multiprocessor. Technical report, The MIT Press, Cambridge, Massachusetts, 1989.
146. S. Kim and J. Browne. A general approach to mapping of parallel computations upon multiprocessor architectures. In *Proc. of Int. Conf. on Parallel Processing*, volume 2, pages 1–8, Aug. 1998.
147. Torben Hagerup. Allocating independent tasks to parallel processors: an experimental study. *J. Parallel Distrib. Comput.*, 47(2):185–197, 1997.
148. Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn. Factoring: a method for scheduling parallel loops. *Commun. ACM*, 35(8):90–101, 1992.
149. C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Trans. Comput.*, 36(12):1425–1439, 1987.

150. Ten H. Tzen and Lionel M. Ni. Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):87–98, 1993.
151. Steven Lucco. A dynamic scheduling method for irregular parallel programs. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 200–211. ACM Press, 1992.
152. H.A. James, K.A. Hawick, and P.D. Coddington. Scheduling independent tasks on metacomputing systems. In *Parallel and Distributed Computing Systems (PDCS'99),Fort Lauderdale*, Aug. 1999.
153. Veridian Systems. PBS: The Portable Batch System. http://www.openpbs.org.
154. IBM Corporation. *Using and Administering LoadLeveler – Release 3.0*, 4 edition, August 1996. Document Number SC23-3989-00.
155. M. J. Litzkow, M. Livny, and M. W. Mutka. Condor : A hunter of idle workstations. In *8th International Conference on Distributed Computing Systems*, pages 104–111, Washington, D.C., USA, June 1988. IEEE Computer Society Press.
156. R. Armstrong, D. Hensgen, and T. Kidd. The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions, 1998.
157. R.F. Freund, M. Gherrity, S. Ambrosius, M. Campbell, M. Halderman, D. Hensgen, E. Keith, T. Kidd, M. Kussow, J.D. Lima, F. Mirabile, L. Moore, B. Rust, and H.J. Siegel. Scheduling resources in multi-user, heterogeneous, computing environments with smartnet. In *Seventh Heterogeneous Computing Workshop, Orlando, Florida*, March 1998.
158. L.Wang, H.J. Siegel, V.P. Roychowdhury, and A.A. Maciejewski. Task matching and scheduling in heterogeneous computing environments using a geneticalgorithm -based approach. *J. Parallel and Distributed Computing*, 47(1):1–15, Nov. 1997.
159. Tracy D. Braun, Howard Jay Siegel, Noah Beck, Ladislau L. Blni, Albert I. Reuther, Mitchell D. Theys, Bin Yao, Richard F. Freund, Muthucumaru Maheswaran, James P. Robertson, and Debra Hensgen. A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems. In *Proceedings of the Eighth Heterogeneous Computing Workshop*, page 15. IEEE Computer Society, 1999.
160. M. Coli and P. Palazzari. Real time pipelined system design through simulated annealing, 1996.
161. P. Shroff, D. Watson, N. Flann, and R. Freund. Genetic simulated annealing for scheduling datadependent tasks in heterogeneous environments. In *5th IEEE Heterogeneous Computing Workshop (HCW '96)*, April 1996.
162. I. De Falco, R. Del Balio, E. Tarantino, and R. Vaccaro. Improving search by incorporating evolution principles in parallel tabu search. In *1994 IEEE Conference on Evolutionary Computation*, volume 2, pages 823–828, 1994.
163. K. Chow and B. Liu. On mapping signal processing algorithms to a heterogeneous multiprocessor system. In *1991 International Conference on Acoustics, Speech, and Signal Processing - ICASSP 91*, pages 1585–1588, May 1991.
164. Heather Kreger. Web services conceptual architecture (wsca 1.0). Prepared for Sun Microsystems, Inc., IBM Software Group, http://www-4.ibm.com/software/solutions/webservices/pdf/WSCA.pdf+.
165. Frank Leymann. Web Services Flow Language. available from http://www-306.ibm.com/software/solutions/webservices/pdf/WSFL.pdf, May 2001.

166. Sriram Krishnan, Patrick Wagstrom, and Gregor von Laszewski. GSFL: A Workflow Framework for Grid Services. In *Preprint ANL/MCS-P980-0802*, Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL 60439, U.S.A., 2002.
167. S. Thatte. XLANG, Web Services for Business Process Design. Available from http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm, 2001. Specification, Microsoft Corporation.
168. Tony Andrews et al. *Business Process Execution Language for Web Services.* 2nd public draft release, Version 1.1, May 2003.
169. Jia Yu and Rajkumar Buyya. A Novel Architecture for Realizing Grid Workflow using Tuple Spaces. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (GRID 2004), Pittsburgh, USA*, Los Alamitos, CA, USA, Nov. 8 2004. IEEE Computer Society Press.
170. Rajkumar Buyya and Srikumar Venugopal. The Gridbus Toolkit for Service Oriented Grid and Utility Computing: An Overview and Status Report. In *Proceedings of the First IEEE International Workshop on Grid Economics and Business Models (GECON 2004), Seoul, Korea*, pages 19–36. IEEE Press, April 23 2004.
171. Kaizar Amin, Mihael Hategan, Gregor von Laszewski, Nestor J. Zaluzec, Shawn Hampton, and Al Rossi. GridAnt: A Client-Controllable Grid Workflow System. In *37th Hawai'i International Conference on System Science*, Island of Hawaii, Big Island, 5-8 January 2004.
172. http://ant.apache.org/.
173. Elliotte Rusty Harold. *XML: extensible markup language.* IDG Books, San Mateo, CA, USA, 1998.
174. The Condor Team. Dagman (directed acyclic graph manager), 2003. http://www.cs.wisc.edu/condor/dagman/.
175. Jun Qin Thomas Fahringer and Stefan Hainzer. Specification of Grid Workflow Applications with AGWL: An Abstract Grid Workflow Language. In *Proceedings of IEEE International Symposium on Cluster Computing and the Grid 2005 (CCGrid 2005)*, Cardiff, UK, May 9-12 2005. IEEE Computer Society Press.
176. T. Fahringer, R. Prodan, R. Duan, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, A. Villazon, and M. Wieczorek. ASKALON: A Grid Application Development and Computing Environment. In *6th International Workshop on Grid Computing (Grid 2005)*, Seattle, USA, November 2005. IEEE Computer Society Press.
177. Jia Yu and Rajkumar Buyya. A Taxonomy of Workflow Management Systems for Grid Computing. Technical Report GRIDS-TR-2005-1, Grid Computing and Distributed Systems Laboratory, University of Melbourne, Australia, March 10 2005.
178. Object Management Group. Unified Modeling Language (UML). http://www.uml.org/. last acces date: June 15 2005.
179. C. A. Petri. *Kommunikation mit Automaten.* PhD thesis, Institut für Instrumentelle Mathematik, Bonn, 1962.
180. Rik Eshuis and Roel Wieringa. Comparing Petri Net and Activity Diagram Variants for Workflow Modelling - A Quest for Reactive Petri Nets. *Lecture Notes in Computer Science*, 2472:321–351, Nov 2003.
181. Junwei Cao, Stephen A. Jarvis, Subhash Saini, and Graham R. Nudd. Gridflow: Workflow management for grid computing. In *CCGRID '03: Proceedings of the*

3st International Symposium on Cluster Computing and the Grid, page 198, Washington, DC, USA, 2003. IEEE Computer Society.

182. Lerina Aversano, Aniello Cimitile, and Pierpaolo Gallucciand Maria Luisa Villani. FlowManager: A Workflow Management System Based on Petri Nets. In IEEE, editor, *26th Annual International Computer Software and Applications Conference, Oxford, England*, pages 1054–1059, Aug. 2002.
183. H. M. W. (Eric) Verbeek, Alexander Hirnschall, and Wil M. P. van der Aalst. XRL/Flower: Supporting Inter-organizational Workflows Using XML/Petri-Net Technology. In *CAiSE '02/ WES '02: Revised Papers from the International Workshop on Web Services, E-Business, and the Semantic Web*, pages 93–108, London, UK, 2002. Springer-Verlag.
184. S. Pllana, T. Fahringer, J. Testori, S. Benkner, and I. Brandic. Towards an UML Based Graphical Representation of Grid Workflow Applications. In *The 2nd European Across Grids Conference*, Nicosia, Cyprus, January 2004. Springer-Verlag.
185. Ian Taylor, Matthew Shields, and Ian Wang. Resource Management for the Triana Peer-to-Peer Services. In Jarek Nabrzyski, Jennifer M. Schopf, and Jan Węglarz, editors, *Grid Resource Management*, pages 451–462. Kluwer Academic Publishers, 2004.
186. Triana Project. http://www.trianacode.org/. last acces date: June 15 2005.
187. Sriram Krishnan, Randall Bramley, Dennis Gannon, Madhusudhan Govindaraju, Rahul Indurkar, Aleksander Slominski, Benjamin Temko, Richard Alkire, Timothy Drews, Eric Webb, and Jay Alameda. The XCAT Science Portal. In *Proceedings of SC2001*, November 10-16 2001.
188. Robert D. Stevens, Alan J. Robinson, and Carole A. Goble. myGrid: personalised bioinformatics on the information grid. *Bioinformatics*, 19:i302–i304, 2003.
189. IT Innovation Workflow Enactment Engine. http://www.it-innovation.soton.ac.uk/mygrid/workflow/. last acces date: June 07 2005.
190. T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver adn K. Glover, M.R. Pocock, A. Wipat, and P. Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.
191. Taverna Team. Taverna. http://taverna.sourceforge.net/.
192. Dietmar W. Erwin and David F. Snelling. UNICORE: A Grid computing environment. *Lecture Notes in Computer Science*, 2150, 2001.
193. Jim Almond and Dave Snelling. Unicore: Secure and uniform access to distributed resources via the world wide web. White paper, October 1998.
194. E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, K. Blackburn, A. Lazzarini, A. Arbree, R. Cavanaugh, and S. Koranda. Mapping abstract complex workflows onto grid environments. *Journal of Grid Computing*, 1(1):25–39, 2003.
195. I. Foster, J.-S. Vöckler, M. Wilde, and Y. Zhao. Chimera: A Virtual Data System for Representing, Querying and Automating Data Derivation. In *14th International Conference on Scientific Database Management*, Edinburgh, 2002.
196. GriPhyN - Grid Physics Network. http://www.griphyn.org/. last acces date: June 01 2005.
197. Jim Blythe, Ewa Deelman, Yolanda Gil, Carl Kesselman, Amit Agarwal, Gaurang Mehta, and Karan Vahi. The role of planning in grid computing. In *13th*

International Conference on Automated Planning and Scheduling (ICAPS), Trento, Italy, 2003.

198. Junwei Cao, Stephen A. Jarvis, and Subhash Saini. Arms: An agent-based resource management system for grid computing. *Scientific Programming*, 10(2):135–148, 2002.
199. S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A directory service for configuring high-performance distributed computations. In *Proceedings of the 6th IEEE Symposium on High-Performance Distributed Computing*, pages 365–375, Portland, OR, 5-8 August 1997.
200. Rich Wolski, Neil T. Spring, and Jim Hayes. The Network Weather Service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6):757–768, 1999.
201. Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, Kent Blackburn, Albert Lazzarini, Adam Arbree, and Scott Koranda. Mapping abstract complex workflows onto grid environments. *Journal of Grid Computing*, 1:25–39, 2003.
202. Jan Weglarz Jarek Nabrzyski, Jennifer Schopf, editor. *Grid Resource Management. State of the Art and Future Trends.* Kluwer Academic Publishers, 2003.
203. Rajesh Raman, Miron Livny, and Marvin H. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *HPDC*, pages 140–, 1998.
204. C. Liu, L. Yang, I. Foster, and D. Angulo. Design and evaluation of a resource selection framework for grid applications. In *In Proceedings of the 11th IEEE Symposium on High-Performance Distributed Computing*, July 2002.
205. Crossgrid. http://www.crossgrid.org/. last acces date: Aug 21 2005.
206. Rajkumar Buyya, David Abramson, and Jonathan Giddy. Nimrod/G: An Architecture of a Resource Management and Scheduling System in a global Computational Grid. In *the 4th International Conference on High-Performance Computing in the Asia-Pacific Region*, pages 283–289. IEEE Press, May 2000.
207. F. Berman and R. Wolski. The AppLeS Project: A Status Report. In *The 8th NEC Research Symposium*, May 1997.
208. Mumtaz Siddiqui and Thomas Fahringer. GridARM: Askalon's Grid Resource Management System. In *Advances in Grid Computing - EGC 2005 - Revised Selected Papers*, volume 3470 of *Lecture Notes in Computer Science*, pages 122–131, Amsterdam, Netherlands, June 2005. Springer Verlag GmbH, ISBN 3-540-26918-5.
209. Mumtaz Siddiqui, Thomas Fahringer, Juergen Hofer, and Ioan Toma. Grid resource ontologies and asymmetric resource-correlation. In German Society of Informatics, editor, *2nd International Conference on Grid Service Engineering and Management (GSEM'05)*, Erfurt, Germany, September 19-22 2005. Lecture Notes in Informatics.
210. Klaus Krauter, Rajkumar Buyya, and Muthucumaru Maheswaran. A taxonomy and survey of grid resource management systems for distributed computing. *Software–Practive and Experience*, 32:135–164, 2002.
211. Bill Allcock, Joe Bester, John Bresnahan, Ann L. Chervenak, Ian Foster, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, and Steven Tuecke. Data management and transfer in high-performance computational grid environments. *Parallel Computing*, 28(5):749–771, May 2002.

212. R. Milner. The polyadic pi-calculus: a tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 203–246. Springer-Verlag, 1993.