Programming and Parallelising Applications for Distributed Infrastructures

Enric Tejedor

Advisor:
Rosa M. Badia

A dissertation submitted in partial fulfillment of
the requirements for the degree of:

Doctor per la Universitat Politècnica de Catalunya

Doctorat en Arquitectura de Computadors
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya

Barcelona
June 2013

A la Pasqui i l'Enric.
Per ser incondicionals.

Abstract

The last decade has witnessed unprecedented changes in parallel and distributed infrastructures. Due to the diminished gains in processor performance from increasing clock frequency, manufacturers have moved from uniprocessor architectures to multicores; as a result, clusters of computers have incorporated such new CPU designs. Furthermore, the ever-growing need of scientific applications for computing and storage capabilities has motivated the appearance of grids: geographically-distributed, multi-domain infrastructures based on sharing of resources to accomplish large and complex tasks. More recently, clouds have emerged by combining virtualisation technologies, service-orientation and business models to deliver IT resources on demand over the Internet.

The size and complexity of these new infrastructures poses a challenge for programmers to exploit them. On the one hand, some of the difficulties are inherent to concurrent and distributed programming themselves, e.g. dealing with thread creation and synchronisation, messaging, data partitioning and transfer, etc. On the other hand, other issues are related to the singularities of each scenario, like the heterogeneity of Grid middleware and resources or the risk of vendor lock-in when writing an application for a particular Cloud provider.

In the face of such a challenge, programming productivity - understood as a tradeoff between programmability and performance - has become crucial for software developers. There is a strong need for high-productivity programming models and languages, which should provide simple means for writing parallel and distributed applications that can run on current infrastructures without sacrificing performance.

In that sense, this thesis contributes with Java StarSs, a programming model and runtime system for developing and parallelising Java applications on distributed infrastructures. The model has two key features: first, the user programs in a fully-sequential standard-Java fashion - no parallel construct, API call or pragma must be included in the application code; second, it is completely infrastructure-unaware, i.e. programs do not contain any details about deployment or resource management, so that the same application can run in different infrastructures with no changes. The only requirement for the user is to select the application tasks, which are the model's unit of parallelism. Tasks can be either regular Java methods or web service operations, and they can handle any data type supported by the Java language, namely files, objects, arrays and primitives.

For the sake of simplicity of the model, Java StarSs shifts the burden of parallelisation from the programmer to the runtime system. The runtime is responsible from modifying the original application to make it create asynchronous tasks and synchronise data accesses from the main program. Moreover, the implicit inter-task concurrency is automatically found as the application executes, thanks to a data dependency detection mechanism that integrates all the Java data types.

This thesis provides a fairly comprehensive evaluation of Java StarSs on three different distributed scenarios: Grid, Cluster and Cloud. For each of them, a runtime system was designed and implemented to exploit their particular characteristics as well as to address their issues, while keeping the infrastructure unawareness of the programming model. The evaluation compares Java StarSs against state-of-the-art solutions, both in terms of programmability and performance, and demonstrates how the model can bring remarkable productivity to programmers of parallel distributed applications.

Acknowledgements

This thesis has been supported by the following institutions: the Universitat Politècnica de Catalunya with a UPC Recerca predoctoral grant; the Spanish Ministry of Science and Innovation and the Comisión Interministerial de Ciencia y Tecnología (CICYT), with contracts TIN2007-60625 and CSD2007-00050; the European Commission in the context of the HiPEAC Network of Excellence (contract IST-004408), the HPC-Europa2 Research Infrastructure (contract 222398), the FP6 CoreGRID Network of Excellence (contract IST-2002-004265), the FP6 XtreemOS project (contract IST- FP6-033576) and the FP7 OPTIMIS project (grant agreement 257115); the Generalitat de Catalunya (contract 2009-SGR-980 and travel grant BE-DGR 2009).

Contents

Abstract **ii**

Acknowledgements **iii**

Contents **x**

List of Figures **xvii**

List of Tables **xix**

1 Introduction **1**
1.1 Context and Motivation 1
1.1.1 Evolution in Parallel and Distributed Infrastructures . . . 1
1.1.2 The Programming Productivity Challenge 2
1.1.3 Approaches to Parallelism and Distribution 3
1.1.4 StarSs for Parallel and Distributed Infrastructures 4
1.2 Contributions 6
1.2.1 Parallel Programming Model for Java Applications 7
1.2.2 Runtime System for Distributed Parallel Infrastructures . 8
1.2.2.1 Grid 9
1.2.2.2 Cluster 9
1.2.2.3 Cloud 9
1.3 Thesis Organisation 10

2 Programming Model **13**
2.1 Overview 14
2.1.1 Basic Steps 14
2.1.1.1 Identifying the Potential Tasks 14
2.1.1.2 Defining a Task Selection Interface 15
2.1.2 Sequential Programming 16
2.2 The Task Selection Interface 17
2.2.1 Method-level Annotations 17
2.2.1.1 @Method 17
2.2.1.2 @Service 18
2.2.1.3 @Constraints 19

2.2.2 Parameter-level Annotations 19
2.2.2.1 @Parameter 19
2.3 The Main Program 20
2.3.1 Scenarios 21
2.3.1.1 Regular Application 21
2.3.1.2 Composite Service 21
2.3.2 Invoking Tasks 22
2.3.2.1 Methods 22
2.3.2.2 Services 23
2.3.3 Sharing Data Between Tasks 24
2.3.4 Working with Objects 24
2.3.4.1 Objects in a Task 24
2.3.4.2 Access in Main Program 26
2.3.5 Working with Arrays 27
2.3.5.1 Arrays in a Task 28
2.3.5.2 Access in Main Program 28
2.3.6 Working with Primitive Types 28
2.3.6.1 Primitives in a Task 29
2.3.6.2 Access in Main Program 29
2.3.7 Working with Files 29
2.3.7.1 Files in a Task 30
2.3.7.2 Access in Main Program 31
2.4 Summary 31

3 Runtime: Common Features **33**
3.1 General Structure 33
3.2 Bytecode Instrumentation 35
3.2.1 How? 35
3.2.2 When? 36
3.2.3 What? 36
3.2.4 What For? 37
3.2.4.1 Asynchronous Task Generation 37
3.2.4.2 Data Access Surveillance 38
3.3 Data Dependency Analysis 38
3.4 Data Renaming 41
3.5 Data Layout and Transfer 41
3.6 Task Scheduling 42
3.6.1 Method Tasks 43
3.6.1.1 Algorithms 43
3.6.1.2 Pre-scheduling 44
3.6.2 Service Tasks 44
3.7 Task Submission, Execution and Monitoring 46
3.8 Summary 46

4 Grid **49**

4.1 Context . . . 49
4.1.1 The Grid . . . 49
4.1.1.1 Architecture . . . 50
4.1.1.2 Virtual Organisations . . . 51
4.1.1.3 Secure Access . . . 51
4.1.1.4 Data Management . . . 52
4.1.2 e-Science Applications . . . 52
4.1.3 Grid APIs: Standardisation Efforts . . . 53
4.1.4 Component-Based Grid Software . . . 54
4.2 Runtime Design . . . 55
4.2.1 Componentisation . . . 56
4.2.1.1 Task Analyser . . . 57
4.2.1.2 Task Scheduler . . . 57
4.2.1.3 Job Manager . . . 57
4.2.1.4 File Manager . . . 58
4.2.2 Uniform Grid API . . . 58
4.2.3 Execution Model . . . 58
4.2.4 Data Model . . . 59
4.3 Relevant Technologies . . . 60
4.3.1 ProActive . . . 60
4.3.2 The Grid Application Toolkit . . . 60
4.4 Programmability Evaluation . . . 61
4.4.1 Taverna . . . 61
4.4.2 Hmmpfam Application . . . 62
4.4.3 Comparison . . . 63
4.4.3.1 Hmmpfam in Java StarSs . . . 63
4.4.3.2 Hmmpfam in Taverna . . . 65
4.4.3.3 Discussion . . . 66
4.5 Experiments . . . 67
4.5.1 Large-Scale Tests . . . 67
4.5.1.1 The Discrete Application . . . 67
4.5.1.2 Testbed . . . 69
4.5.1.3 Results . . . 72
4.5.2 Small-Scale Tests . . . 78
4.5.2.1 Component Distribution in Nord . . . 78
4.5.2.2 Hmmpfam in MareNostrum . . . 80
4.6 Related Work . . . 84
4.6.1 Grid Programming Models . . . 84
4.6.2 Workflow Managers . . . 85
4.6.3 Component-Based Grid Software . . . 86
4.7 Summary . . . 86

5 Cluster **89**
5.1 Context . . . 89
5.1.1 Cluster Computing . . . 89
5.1.2 Cluster versus Grid . . . 90
5.1.3 Productivity in Cluster Programming: APGAS . . . 91
5.2 Runtime Design . . . 92
5.2.1 Java StarSs and APGAS . . . 92
5.2.2 Runtime Structure . . . 93
5.2.3 Communication Protocol . . . 94
5.2.4 Execution Model . . . 95
5.2.5 Data Model . . . 95
5.2.5.1 Data Layout . . . 95
5.2.5.2 Data Transfer . . . 96
5.2.5.3 Data Reuse and Locality . . . 96
5.3 Relevant Technologies . . . 96
5.3.1 IBM APGAS Runtime . . . 96
5.4 Programmability Evaluation . . . 97
5.4.1 The X10 Programming Language . . . 97
5.4.1.1 Places and Activities . . . 98
5.4.1.2 Synchronisation . . . 98
5.4.1.3 Data Distribution . . . 98
5.4.2 Application Description . . . 98
5.4.2.1 Matrix Multiplication . . . 99
5.4.2.2 Sparse LU . . . 102
5.4.2.3 K-means . . . 104
5.4.3 Programmability Discussion . . . 105
5.5 Experiments . . . 107
5.5.1 Testbed . . . 107
5.5.2 X10 Comparison Results . . . 107
5.5.2.1 Test Setup . . . 107
5.5.2.2 Matrix Multiplication . . . 108
5.5.2.3 Sparse LU . . . 110
5.5.2.4 K-means . . . 112
5.5.3 NAS Parallel Benchmarks . . . 115
5.5.3.1 Test Setup . . . 115
5.5.3.2 Embarrassingly Parallel (EP) . . . 117
5.5.3.3 Fourier Transformation (FT) . . . 117
5.5.3.4 Integer Sort (IS) . . . 117
5.6 Related Work . . . 118
5.7 Summary . . . 119

6 Cloud **121**
6.1 Context . . . 121
6.1.1 Cloud Computing . . . 121
6.1.2 Clouds and Service-Oriented Architectures . . . 123
6.1.3 Clouds for HPC Science . . . 124

6.2 Runtime Design . . . 125
6.2.1 Support for Services as Tasks . . . 125
6.2.2 Integration In a Service-Oriented Platform . . . 126
6.2.3 Exploitation of Virtual Cloud Resources . . . 127
6.3 Relevant Technologies . . . 128
6.3.1 Cloud Provider Connectors . . . 129
6.3.2 SSH Adaptor of JavaGAT . . . 129
6.3.3 Apache CXF . . . 129
6.4 Programmability Evaluation . . . 129
6.4.1 Programming with Services . . . 130
6.4.1.1 WS-BPEL . . . 130
6.4.1.2 Travel Booking Service . . . 130
6.4.1.3 Comparison . . . 132
6.4.2 Programming with Objects . . . 134
6.4.2.1 Deployment . . . 134
6.4.2.2 Object Creation . . . 134
6.4.2.3 Asynchronous Computations . . . 135
6.4.2.4 Synchronisation . . . 137
6.4.2.5 Termination . . . 137
6.5 Experiments . . . 138
6.5.1 Gene Detection Composite . . . 138
6.5.2 Testbed . . . 139
6.5.3 Resource Elasticity and Cloud Bursting . . . 141
6.5.4 Performance . . . 144
6.6 Related Work . . . 148
6.6.1 Platform-as-a-Service Solutions . . . 148
6.6.2 Frameworks for Service Composition . . . 149
6.6.3 Cloud Programming Models . . . 150
6.7 Summary . . . 151

7 Conclusions and Future Work **153**
7.1 Programming Model . . . 153
7.1.1 Future work . . . 155
7.2 Runtime System . . . 155
7.2.1 Future work . . . 157

Bibliography **159**

A Applications **173**
A.1 Hmmpfam - Java StarSs . . . 173
A.2 Discrete - Java StarSs . . . 175
A.2.1 Main Program . . . 175
A.2.2 Task Selection Interface . . . 176
A.2.3 Task Graph . . . 178
A.3 Gene Detection - Java StarSs . . . 179
A.3.1 Main Program . . . 179

A.3.2 Task Selection Interface 180

B Resource Description **183**

B.1 Resources File 183

B.2 Project File 185

List of Figures

1.1 Star Superscalar execution model. 5
1.2 Thesis organisation. 10

2.1 Steps of the Java StarSs programming model. In the application, which is programmed sequentially, the user identifies the methods and services to be tasks and then selects them. The model is based on inter-task parallelism and task asynchrony. 15
2.2 Example of code refactoring. An application that increments the rows of a matrix (a), composed by two loops, is reorganised to encapsulate the inner loop in a method incrementRow (b) so that it can become a task. 16
2.3 Syntax of a task selection interface, comprised of a method task and a service task declarations. The annotations are represented in bold: @Method for identifying a method, @Service for a service operation, @Constraints to specify the resource requirements of a method task and @Parameter to state the direction and type of a method task parameter. The elements of each annotation are in italics. 18
2.4 Parts of the application code: main program and task code. In the main program, except for the black-box area, the programming model features are enabled. 21
2.5 Examples of two scenarios for a Java StarSs application: (a) regular application with a main method that starts the execution, (b) composite service operation provided by a remotely-accessible web service. 22
2.6 Examples of task invocations from a main program (a), for both methods and services (including stateless and stateful-like invocations). In (b), the corresponding task selection interface is shown as a complement to the main program in (a). 23
2.7 Sample sequential application (a) and its corresponding task selection interface (b). Method foo, implemented by class example.A, is chosen as a task in (b); it is an instance method (invoked on an object of class A), receives an input object parameter of class B and returns a C object. 25

2.8 Case of synchronisation by transition to black-box area. Object y is returned by method task of class X, which we assume was selected as a task and therefore is spawned asynchronously. When the main program reaches the call to method blackBox, which is implemented in a non-watched class Z, a synchronisation takes place to get y and pass it to blackBox. 27

2.9 Example of synchronisation by access to an array element from the main program. foo is assumed to be a task that receives a one-dimensional array as a parameter, updates it and returns a two-dimensional array. Those arrays are accessed later in the main program, each forcing a synchronisation. 28

2.10 Example of synchronisation for primitive types. The invocation of the task method foo is synchronous here, because of the integer value i that it returns. Primitive types that are passed as task parameters, like b, do not require synchronisation. 29

2.11 Sample sequential application (a) and its task selection interface (b). Method increment is chosen as a task in (b); it receives an input/output file parameter where a counter value is stored and increments that value. In (a), the main program opens streams on the same file incremented by the task. 30

3.1 Java StarSs runtime structure. The master side deals with the main program of the application, whereas the worker side handles the task code. 34

3.2 As a result of applying the programming model, the user provides the Java classes corresponding to the task selection interface and the sequential application. In order to enable its parallelisation, the application is instrumented to insert calls to the Java StarSs runtime at certain key points. At execution time, the runtime will use the information in the interface to parallelise the instrumented application. 35

3.3 Main program of the Sum application (a), its corresponding task selection interface (b) and the graph generated when running it (c). At every iteration, the genRandom task method generates a random number and writes it in file rdFile; after that, method add (also a task) adds that number to a sum stored in the sum object. When executing the application, the runtime detects different kinds of dependencies, some of which can be avoided by means of a data renaming technique (WaW, WaR), whereas some cannot (RaW). 40

3.4 In the code snippet in (a), A is a matrix divided in $N \times N$ blocks. The createBlock method allocates a single block of size $M \times M$ doubles and initialises all its elements with a given constant VAL. createBlock is selected as a task in the interface in (b) and is also marked as an initialisation task (isInit = true field in the @Method annotation); note that the parameters of createBlock do not need the @Parameter annotation, since their type is primitive and, consequently, their direction is IN. Finally, the scheduling of the createBlock initialisation tasks leads to the allocation of blocks among resources shown in (c), assuming 3 resources, 4 slots per resource and N=6. 45
3.5 Overview of the basic features of the Java StarSs runtime. 47

4.1 Grid architecture layers. Courtesy of the GridCafé website [23]. . 50
4.2 Location of the Java StarSs runtime in the Grid stack. 56
4.3 Component hierarchy and interactions in the Grid Java StarSs runtime, which sits on top of a uniform Grid API. 57
4.4 Simple workflow in Taverna. Node1 has two input ports and two output ports, while Node2 has only one of each kind. The link between the two nodes represents a data dependency. 62
4.5 Task selection interface corresponding to the Hmmpfam application in Java StarSs. 64
4.6 Example of a task dependency graph generated by Hmmpfam when running it with Java StarSs. In this case, the database is split in two fragments and the query sequences file in four parts. This creates eight independent tasks that run hmmpfam on a pair of database-sequence fragments. After that, there are three levels of reduction tasks, the last one merging the results from the two different database fragments. 64
4.7 First version of Hmmpfam in Taverna. 65
4.8 Second version of Hmmpfam in Taverna. 66
4.9 Testbed comprising two large-scale scientific grids (Open Science Grid, Ibergrid) and a local BSC-owned grid. The Discrete application, running on a laptop with Java StarSs, interacts with the grids through GAT and its middleware adaptors. 71
4.10 Test results for the Discrete application when run with Java StarSs in the Grid testbed: (a) distribution of the Discrete tasks among the three grids; (b) comparison of percentage of transfers between the locality-aware and FIFO scheduling algorithms; (c) evolution of the number of transfers when applying locality-aware scheduling. 74
4.11 Detail of the task constraint specification for the Discrete application. The complete task selection interface can be found in Appendix A.2.2. 77

4.12 Reduced version of the Discrete graph, only for illustrative purposes (the real one is in Appendix A.2.3). The constraints in Figure 4.11 lead to the task scheduling on the three grids represented by this figure. 77
4.13 Deployments of the Mergesort runtime: Single-node and Distributed. 79
4.14 Performance comparison for Hmmpfam between Java StarSs and MPI-HMMER. 81
4.15 Execution of Hmmpfam with Java StarSs. The figure depicts the percentage of Idle+Transferring time in the workers, with respect to the total of Idle+Transferring+Computing, with and without pre-scheduling. 82
4.16 Number of concurrent transfers that Java StarSs is performing during the first 500 seconds of Hmmpfam, varying the number of worker cores (16, 64, 256) and applying pre-scheduling or not. Pre-scheduling keeps the master busy (transferring) longer, except in case of overload. 83

5.1 Design of Java StarSs on top of APGAS. 93
5.2 Pseudo-code representing the skeleton of the Java StarSs runtime that is run in all nodes. Essentially, the main node executes the main program of the application and the worker nodes wait to respond to incoming AMs. 94
5.3 Cluster Java StarSs architecture: Java StarSs runtime on top of the APGAS runtime, invoking the latter through Java bindings. X10 shares the same underlying APGAS layer as Java StarSs. . . 97
5.4 Main algorithm of the matrix multiplication application in Java StarSs. The method multiply multiplies two input blocks of matrices A and B and accumulates the result in an in-out block of matrix C. 99
5.5 Implementation in X10 of the matrix multiplication benchmark. (a) contains the creation, initialisation and distribution of the three matrices A, B and C involved in the computation. (b) shows the main algorithm. 100
5.6 A second implementation of the X10 matrix multiplication. In this version, the three matrices created in (a) are distributed. The main algorithm is not shown since it is equivalent to the one in Figure 5.5(b). The fact of distributing matrices A and B makes necessary to add some code, depicted in (b), to the activity method multiply for explicitly transferring blocks. 101
5.7 X10 matrix distributions used in the tested benchmarks: (a) Block distribution along the 0th axis, (b) Block distribution along the 1st axis, (c) Block Cyclic distribution along the 0th axis with a block size of two rows. In the benchmarks, each cell of a distributed matrix is itself a sub-matrix (i.e. a block of the benchmark). 102

5.8 (a) Main algorithm of the Sparse LU benchmark for Java StarSs and (b) the corresponding task dependency graph generated for an input matrix of 5x5 blocks. Different node colours in (b) represent different task methods and the number in each node is the generation order. Also in (b), the three highlighted task levels correspond to the three different finish blocks in the X10 implementation. 103

5.9 Test results for the Matrix multiplication benchmark for Java StarSs and X10. Study of the best block size, with a fixed number of 64 cores, keeping the same problem size and varying the block size: (a) benchmark execution times and (b) average task/activity times. Scalability analysis: (c) execution times and (d) speedup for a range of cores, input matrices of N=64 and M=200, i.e. 64x64 blocks of size 200x200 doubles; for X10, two different configurations of the matrices are considered: replicating matrices A and B (ABRep) or distributing them (ABDist). In (e), study of different problem sizes with a fixed number of 64 cores and using the best block size found (200x200). 109

5.10 Test results for the Sparse LU benchmark for Java StarSs and X10. Study of the best block size, with a fixed number of 64 cores, keeping the same problem size and varying the block size: (a) benchmark execution times and (b) average task/activity times. Scalability analysis: (c) execution times and (d) speedup for a range of cores, input matrices of N=64 and M=300, i.e. 64x64 blocks of size 300x300 doubles; for X10, two different partitionings of the matrix to factorise are considered: Block distribution and Block Cyclic distribution. In (e), study of different problem sizes with a fixed number of 64 cores and using the best block size found (300x300). 111

5.11 Test results for the K-means application for Java StarSs and X10. Study of the best fragment size, with a fixed number of 64 cores, keeping the same problem size and varying the fragment size: (a) application execution times and (b) average task/activity times. Scalability analysis: (c) execution times and (d) speedup for a range of cores, input parameters: 128000000 points, 4 dimensions, 512 clusters, 50 iterations; two fragment sizes are considered: 31250 points and 500000 points. (e) influence of JIT compilation in the iteration time for the two fragment sizes. In (f), study of different problem sizes with a fixed number of 64 cores and using the best fragment sizes found (31250 for Java StarSs, 500000 for X10). 114

5.12 Execution times (seconds) of the NAS parallel benchmarks: (a) Embarrassingly Parallel, (b) Fourier Transformation and (c) Integer Sort. Tested implementations: Java StarSs, ProActive, F-MPJ and NPB-MPI (original). 116

6.1 Location of the Java StarSs programming model, runtime and applications in the Cloud stack. 123
6.2 Architecture of the Java StarSs Cloud runtime. A service hosted in a Web services container can be accessed by any service consumer (e.g. web portal, application). The interface of this service offers several operations, which can be composites previously written by a service developer following the Java StarSs programming model. When the container receives a request for a given composite, the Java StarSs runtime starts generating the corresponding task dependency graph on the fly, so that it can orchestrate the execution of the selected tasks. Service tasks will lead to the invocation of external services (possibly deployed in the Cloud), while method tasks can be run either on virtualised Cloud resources or on physical ones. 126
6.3 Technologies leveraged by the Java StarSs Cloud runtime. 128
6.4 In (a), graphical workflow of the travel booking composite, as shown by the Eclipse BPEL Designer; the invocations to external services are numbered. In (b), a fragment of the corresponding WS-BPEL document, focusing on the invocation of service BookFlight. 131
6.5 Java StarSs version of the travel booking composite service: (a) main program of the composite and (b) task selection interface. In (a), the calls to external services are underlined. 133
6.6 Java StarSs version of N-body: (a) main program and (b) task selection interface. 135
6.7 Comparison of key fragments in the N-body application. 136
6.8 Task dependency graph generated for N-body, with a universe of 3 domains and 3 iterations. Yellow (light) tasks correspond to the addForce method, whereas red (dark) ones represent calls to moveBody. 137
6.9 Gene detection composite service. The dependency graph of the whole orchestration is depicted on the right of the figure: circles correspond to method tasks and diamonds map to service task invocations, while stars represent synchronisations due to accesses on task result values from the main program. A snippet of the composite code is provided, focusing on a particular fragment which runs BLAST to obtain a list of sequences and then parses their identifiers. The graph section generated by this piece of code is also highlighted in the overall structure of the composite. 139
6.10 Testbed comprising two clouds: a private cloud, located at BSC, and the Amazon EC2 public cloud (Ireland data centre). The GeneDetection composite service is deployed in a server machine, which contacts the VMs of the private cloud through a VPN. An external server publishes the operations corresponding to service tasks. 140

6.11 Execution of two requests for the gene detection composite that illustrates the elasticity and bursting features of the Java StarSs runtime: (a) evolution of the load generated by the composite's method tasks; (b) evolution of the number of VMs in the private cloud and Amazon EC2; (c) state of the VMs during the execution of the requests. 142
6.12 Graph generated by the GeneWise computation in the gene detection composite, for an execution that finds 8 relevant regions in the genomic sequence. Red (dark) tasks correspond to the genewise method, whereas yellow (light) ones represent calls to mergeGenewise. 144
6.13 Execution of the GeneWise computation, with private VMs only and bursting to Amazon: (a) evolution of the number of tasks, (b) VM elasticity. 145
6.14 Execution times of the GeneWise computation, with private VMs only ('Private') and a combination of private and public VMs ('Hybrid'). 147

A.1 Main program of the Hmmpfam application for Java StarSs. . . . 174
A.2 Main program of the Discrete application for Java StarSs. 176
A.3 Task selection interface of the Discrete application for Java StarSs. 177
A.4 Graph generated by Java StarSs for Discrete; input parameters: 10 structures, 27 different configurations of EPS, FSOLV and FVDW. 178
A.5 Main program of the Gene Detection composite for Java StarSs. 180
A.6 Task selection interface of the Gene Detection composite for Java StarSs. 181

B.1 Snippet of a resources file. 185
B.2 Snippet of a project file. 186

List of Tables

4.1 Job submission and file transfer statistics for Discrete. 75
4.2 Influence of component distribution in Mergesort 80

5.1 Number of code lines of the tested applications. 106

6.1 Statistics of the GeneWise part of the gene detection composite. Times in seconds. 146

Chapter 1

Introduction

1.1 Context and Motivation

1.1.1 Evolution in Parallel and Distributed Infrastructures

The last decade has witnessed unprecedented changes in parallel and distributed infrastructures. The year 2002 marked a turning point in the field of computer architecture, when improving processor performance by increasing clock frequency became hardly sustainable, due to three main factors: the growing gap between processor and memory speeds (the memory wall), the stalemate in the exploitation of instruction-level parallelism (the ILP wall) and the limitations in power dissipation (the power wall) [83]. As a response to those issues, manufacturers engaged in the design of *multicore architectures*, which gather simple processing units on the same die. The current fastest supercomputers are based on multicore technologies, sometimes combining them with specialised GPUs (Graphics Processing Units) in hybrid setups. Moreover, the fact that multicores are available at affordable costs has contributed to the *commoditisation of parallel hardware*, making it reach a broader audience.

On the applications side, some scientific programs - from diverse fields like particle physics, bio-informatics or earth sciences - came up with a need for large computing and storage capabilities. This computationally-intensive science working on immense data sets was named *e-Science*, and often required more high-performance computing resources than those of a single institution. As a result, users from different communities started to share their resources to build *grids* [128]: infrastructures that combine loosely-coupled resources from multiple administrative domains scattered over a wide geographic area. Scientists from around the globe can use grids to tackle large and complex tasks, to accomplish projects that would be impossible otherwise. Besides, grids enable global collaboration: scientists can share data, data storage space, computing power, expertise and results in a large scale. A key feature of grids is their *heterogeneity*, not only in terms of the resources they federate, but also regarding the middleware that provides the basic services to access those resources.

In its early days, one of the concepts behind Grid computing was that of *utility computing*: a global public Grid was envisaged to provide on-demand and metered access to computation and data just like the power grid delivers electricity. Although never truly realised for grids, the concept was later revisited by some private companies like Amazon, which in 2006 started offering the Elastic Compute Cloud (EC2) [2]. In addition to utility computing, EC2 relied on *virtualisation* technologies, which had resurged to improve resource usage and management in Internet service providers [115]. Hence, Amazon and other vendors started renting virtual machines - deployed in their datacentres - to customers in a pay-as-you-go basis; moreover, this was delivered *as a service* over the Internet. Such combination of utility computing, virtualisation and service-orientation was popularised as *Cloud computing*. Nowadays, clouds allow to outsource any part of the IT stack, i.e. not only hardware but also development /deployment platforms and entire applications. Many Cloud providers have appeared so far, each offering its own interface to access its services.

1.1.2 The Programming Productivity Challenge

The previous subsection has explained how parallel and distributed infrastructures have increased in size and complexity. On the one hand, multicores have shifted the focus of performance from hardware to software. It is no longer sufficient to write single-threaded code and rely on new CPUs to boost performance; instead, applications are required to manage the concurrent execution of multiple threads to exploit all the cores [140]. This complicates the job of the programmer, who is faced with two main duties: (i) *thinking about parallelism*, which involves identifying the computations that compose the application and the data they share, and sometimes restructuring the program in a way that favours concurrency; (ii) *dealing with parallelisation*, which entails things like creating and synchronising threads, scheduling/balancing the work load between threads, debugging and fixing data races and deadlocks, etc.

Furthermore, the computation/storage demands of an application may require it to execute over a set of *distributed resources*. In this scenario, the aforementioned duties of parallel programming are still present and they can get even more complex (e.g. synchronisation of remote processes, data exchange by message passing). In addition, distributed systems introduce some new concerns. Perhaps one of the biggest is achieving a consistent view of data across processes, since every process works with its own private memory and may update shared data. Another example is fault tolerance: a distributed application spawning processes in different nodes should continue executing properly in the event of a failure, either at process or hardware (node, network) level. Finally, as a distributed infrastructure grows in size, the scalability of the applications that execute on it becomes increasingly relevant.

On the other hand, some of the difficulties met by the programmer are not strictly related to parallel/distributed programming, but to the singularities of each infrastructure. Grids are inherently heterogeneous, both in terms of their resources and the middleware that manages them. Hence, a Grid application

may need to function over machines with different architectures and operating systems. Besides, there is no standard way of accessing Grid services such as job submission and file transfer, and instead that depends on the interface provided by the particular middleware installed in a grid, which hinders portability. A similar problem exists in current clouds: every vendor offers its own API to reserve virtual machines in its infrastructure or to develop applications for its platform, which increases the risk of lock-in when writing an application for a given Cloud provider.

Due to all the factors discussed above, *programming productivity*, understood as a tradeoff between ease of programming and performance, has become crucial [83, 140]. For economic reasons, it is not enough anymore to merely make an efficient use of hardware, it is also necessary now to make a highly efficient use of software developers, whose time is valuable. Moreover, new commodity parallel architectures should be made available to a vast majority of developers that lack concurrent programming expertise. This brings a need for parallel languages and programming models that assist developers when writing applications for parallel and distributed infrastructures. Ideally, the applications developed in those languages/models should be portable, i.e. not tied to a certain platform.

1.1.3 Approaches to Parallelism and Distribution

Different types of approaches have been proposed to achieve parallelism and distribution in applications, each requiring a certain level of effort or control from the programmer [161]. At one extreme, some research in parallelising compilers has been conducted in the past decades, in order to automatically convert sequential programs into a parallel form (mainly focusing on loops). However, the results of that research are still limited, especially for object-oriented languages, due to the complexity of detecting when it is safe or worth to parallelise codes with non-trivial data dependencies [160, 154].

The impossibility to simply rely on a compiler to efficiently parallelise an application made the programming community move on to *explicit parallel programming*. Nevertheless, this is not an easy step because most mainstream languages were designed for sequential programming [95, 94]. The lack of support for concurrency and distribution in those languages was initially compensated with special libraries for threading, synchronisation and remote communication, e.g. Pthreads [50] in C/C++ or RMI [158] in Java. In this category, two models have gained the widest acceptance in high-performance computing: OpenMP [102] and MPI [116]. On the one hand, OpenMP offers an interface to make a master thread fork child threads that work in parallel; it is relatively easy to use compared to raw threading, but it is restricted to shared-memory systems and fork-join parallelism. On the other hand, MPI can work in distributed environments, structuring a computation in parallel processes that exchange messages; nevertheless, it requires a considerable effort and expertise to, for instance, fragment the application data and manage the communication between processes.

Without abandoning explicit parallelism but aiming for better productivity, other approaches integrate concurrency and distribution in the syntax of a language, providing means to express the parallel structure of an application in a higher level; such means include special constructs for e.g. loop parallelisation, message passing, spawning of computations or data distribution. This has been done either by extending an existing mainstream language with special syntax [110, 146] or by creating a brand new language [101, 100, 178]; the first option usually has a lower learning curve, since programmers can reuse their knowledge of the original language and incrementally learn the new syntax. A family of languages in this category is based on the Partitioned Global Address Space (PGAS) model, which presents a shared partitioned address space to simplify the development of distributed applications, while exposing data locality to enhance performance; some of these languages follow a pure SPMD pattern [110, 146], while others are able to dynamically spawn asynchronous threads [101, 100]. Another group in this category is the so-called concurrency-oriented languages [178, 121], which focus on distribution and fault tolerance. The most successful one is perhaps Erlang, which expresses distributed applications as a set of lightweight processes that share nothing and communicate through messages, but in a more natural and easy way than e.g. MPI, by using high-level language constructs that keep the network transparent.

As opposed to the aforementioned examples, *implicit parallel programming models* for distributed-memory machines [184, 103, 84] feature few or no explicit parallel constructs and primitives. Instead, they combine a sequential syntax with parallelism discovered at execution time. Therefore, they shift the parallelisation effort from the user to the implementation of the model, thus making possible for non-expert programmers to produce concurrent codes. In Section 1.1.2 we distinguished between reasoning about parallelism and actually battling with issues related to parallelisation and distribution, as the two main tasks of the programmer. In implicit models, the former is still recommendable to create opportunities for concurrency in the application, but the programmer is freed from the latter and the complexity of the underlying system is hidden. Although implicit models limit the ability of the user to tune for every last bit of performance, they do it to maximise programmability. The tradeoff between these two concepts defines then the productivity delivered by such a model.

1.1.4 StarSs for Parallel and Distributed Infrastructures

Star Superscalar (StarSs) is a task-based and dependency-aware programming model that belongs to the field of implicit parallel programming. Applications in StarSs are developed in a sequential fashion, while a runtime system is in charge of exploiting their intrinsic concurrency at execution time. Parallelism is achieved by means of hints given by the programmer, which identify parts of the code that operate on a set of parameters. Such parts are encapsulated in functions/methods, called tasks. With the help of those hints, task invocations are automatically detected, as well as their data interdependencies. Hence, a dataflow task graph is dynamically generated and tasks are scheduled and run

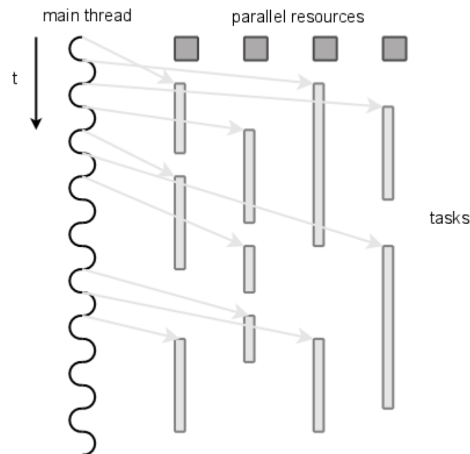


Figure 1.1: Star Superscalar execution model.

in parallel - when dependencies allow - on the available resources. Figure 1.1 illustrates this execution model, where a single thread of control running the main program of the application creates tasks and distributes them on resources.

StarSs is in fact a general term used to design a family of programming models for several hardware platforms, which share the same principles described above. Thus, StarSs has been implemented for Cell/B.E [152], SMP [151] and GPU [155], in each case supported by a specific runtime. This thesis will discuss an implementation of StarSs for parallel distributed infrastructures - such as clusters, grids and clouds - based on the Java language.

The election of Java was motivated by a set of factors. Java is one of the most popular programming languages nowadays, as reported in [51, 64]. Besides, it has several appealing characteristics: object orientation, which favours encapsulation and code reuse; portability, since Java applications are first compiled to an intermediate representation - the bytecode - that can run on any platform provided with a Java Virtual Machine (JVM), which is useful in heterogeneous environments; automatic garbage collection that frees unused memory, which together with strong type checking makes programs more robust. Despite all these benefits, the use of Java in high-performance computing is still limited. The poor performance of the language in its early days - mainly caused by slow bytecode interpretation and garbage collection pauses - hindered its adoption, along with some numerical issues that are not completely solved yet [92, 35]. Nevertheless, the continuous improvements in the Just-in-Time compilers of JVMs, which transform bytecode to native code at execution time, have significantly narrowed the gap between Java's performance and that of languages like C/C++ [165, 82, 162, 1]. Java is extensively used in distributed computing, primarily in software for the Web [8, 32] but also in frameworks for big-data applications [6] and distributed databases [4], or even in contest-winning systems [28]. Regarding HPC, a relevant project based on Java is ESA Gaia [147].

1.2 Contributions

This thesis demonstrates that it is possible to develop a distributed parallel application in a totally sequential fashion and independently of the underlying infrastructure where the application will run.

In that sense, we contribute with (i) the design of an **implicit parallel programming model** for distributed Java applications and (ii) a **runtime system** that implements the features of the aforementioned model for three different distributed parallel infrastructures. With these contributions we address the programming-productivity challenge, trying to maximise the programmability of distributed parallel applications without hindering their performance at execution time.

The publications that support this thesis are listed below in chronological order. In the next subsections we describe the contributions in more detail and link them with the publications.

- [168] E. Tejedor and R. M. Badia, **COMP Superscalar: Bringing GRID superscalar and GCM Together**. In *8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid08)*, May 2008.

- [169] E. Tejedor, R. M. Badia, R. Royo and J. L. Gelpí. **Enabling HMMER for the Grid with COMP Superscalar**. In *10th International Conference on Computational Science 2010 (ICCS10)*, May 2010.

- [171] E. Tejedor, M. Farreras, D. Grove, R. M. Badia, G. Almasi and J. Labarta, **ClusterSs: A Task-Based Programming Model for Clusters**. In *20th International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC11)*, June 2011.

- [173] E. Tejedor, F. Lordan and R. M. Badia, **Exploiting Inherent Task-Based Parallelism in Object-Oriented Programming**. In *12th IEEE/ACM International Conference on Grid Computing (Grid11)*, September 2011.

- [170] E. Tejedor, J. Ejarque, F. Lordan, R. Rafanell, J. Álvarez, D. Lezzi, R. Sirvent and R. M. Badia, **A Cloud-unaware Programming Model for Easy Development of Composite Services**. In *3rd IEEE International Conference on Cloud Computing Technology and Science (CloudCom11)*, November 2011.

- [172] E. Tejedor, M. Farreras, D. Grove, R. M. Badia, G. Almasi and J. Labarta, **A High-Productivity Task-Based Programming Model for Clusters**. In Journal *Concurrency and Computation: Practice and Experience*. Volume 24, Issue 18, pages 2421–2448, December 2012.

1.2.1 Parallel Programming Model for Java Applications

First, this thesis contributes with a parallel programming model for distributed Java applications that will be referred to as **Java StarSs** in this document. The features of this model have been presented in [168], [172], [173] and [170].

Java StarSs targets productivity when developing applications for distributed parallel infrastructures; for that purpose, like other StarSs members, it is based on fully-sequential programming, so that programmers do not need to deal with the typical duties of parallelisation and distribution, such as thread creation and synchronisation, data distribution, messaging or fault tolerance. Besides, Java StarSs incorporates some distinctive properties that are considered contributions of this thesis:

- **Applications are 'clean'**: the model does not require to use any API call, special pragma or construct in the application, everything is pure standard Java syntax and libraries. The hints for parallelisation, like task identification or parameter direction, are provided in an *interface completely separated* from the code of the application. This property brings three benefits: first, it makes possible for some existing applications to be parallelised without any changes to their source code; second, it makes applications *portable* between different distributed infrastructures, since programs do not include any detail that could tie them to a particular platform, like deployment or resource management; third, it facilitates the learning of the model, since Java programmers can reuse most of their previous knowledge of the language.

- **Hybrid tasking model**: like other implicit models, Java StarSs permits to select a method as a task, for it to be spawned on a resource at execution time. In addition, Java StarSs supports tasks that correspond to web service operations, published in some web service container over the Internet. These service tasks are integrated in the dataflow dependency system together with regular method tasks, which means that Java StarSs applications can dynamically generate workflows whose nodes can be either methods or service invocations. In that sense, Java StarSs offers a model to programmatically create *composite services*, that is, applications that reuse functionalities wrapped in services or methods, adding some value to create a new product that can also be published as a service. This model is especially well suited for service-oriented environments like clouds.

- **Complete coverage of data types**: all the data types of the Java language are supported in Java StarSs, both for use in the main program of the application and as task parameters. In addition to files, arrays and primitive types, Java StarSs integrates *objects* in the model. The aim is for the programmer to code as she would do in any sequential Java application, where objects are created, receive invocations and field accesses, are passed as parameters or returned by methods, while the management of concurrency and distribution of these objects is kept transparent to her.

1.2.2 Runtime System for Distributed Parallel Infrastructures

Second, this thesis contributes with a runtime system that has been implemented on top of three different distributed parallel infrastructures: **Grid**, **Cluster** and **Cloud**. The runtime enables the features of the programming model and abstracts it from what is underneath; in order to do so, it needs to handle the peculiarities of each infrastructure while the model is unaware of them. There exist a set of duties that are delegated to the runtime so that the programmer does not have to deal with them, including: bytecode instrumentation, data dependency analysis, data renaming, control of data layout and transfer, task scheduling, task submission, execution and monitoring. Such responsibilities are split between a master part, which runs the main program of the application, and a worker part, which executes the tasks.

Among the functionalities implemented by the Java StarSs runtime, three of them are presented as contributions of this thesis:

- **Instrumentation**: it corresponds to the dynamic transformation of a sequential application into an application that can be parallelised. The instrumentation process basically involves inserting calls to the Java StarSs runtime in the application code before executing it, thus enabling the asynchronous creation of tasks and automatic data synchronisation. This functionality makes possible that the application is completely sequential and eliminates the need for the programmer to use any API.

- **Object management**: the use of objects at the programming model level requires some support in the runtime, in order to coordinate the concurrency and distribution of these objects. Hence, the Java StarSs runtime incorporates the management of objects to the following functionalities: task dependency detection, synchronisation in the main program and data transfer.

- **Orchestration of composite services**: Java StarSs can be used to program composite services, formed by calls to method and service tasks. The Java StarSs runtime is able to orchestrate (steer) the execution of such composites, scheduling and invoking the inner method or service tasks when they are free of dependencies and managing the data exchange between them.

Furthermore, for each infrastructure, this thesis provides an exhaustive study of the **productivity** of Java StarSs, considering two factors: first, *ease of programming*, comparing how a set of representative applications are developed in Java StarSs and in other languages/models of the same field; second, *performance*, presenting *experiments in real-world infrastructures* (e.g. Open Science Grid, MareNostrum supercomputer, Amazon Elastic Compute Cloud) that compare the Java StarSs runtime against other state-of-the-art approaches.

1.2.2.1 Grid

Grids are characterised by their heterogeneity, both in terms of the resources they federate and the middleware that provides the basic services to access those resources. On the one hand, the fact that Java StarSs is based on Java helps **working with heterogeneous resources**, thanks to the portability offered by this language. On the other hand, the Java StarSs runtime for grids is built on top of a uniform API with a set of adaptors, each one implementing the client of a particular grid middleware; this way, the runtime can **interact with grids managed by different middleware**, belonging to different administrative domains and requiring different credentials.

None of the Grid-related details mentioned above appear in the application, thus ensuring that the programming model is not aware of the infrastructure. The work for grids has been published in [168] and [169].

1.2.2.2 Cluster

In order to improve the performance of applications in the Cluster scenario, the Java StarSs runtime for clusters was implemented on top of a communication layer that enables **fast one-sided communications** and the **exploitation of high-speed networks**.

Besides, the design of this runtime incorporates new features for the sake of **scalability**: persistent workers that maintain a cache of in-memory task data, which favours data reuse and locality; data communications between workers, bypassing the master, which reduces the load of the latter; tasks that permit to allocate and initialise data directly in a worker node, so that the total memory is not limited to that of the master node, and so that there is no need to transfer all the data from the master to the workers at the beginning of the application. These new features do not affect the programmability of the model, which is kept as simple as possible. The work for clusters has been published in [171], [172] and [173].

1.2.2.3 Cloud

In its most recent version, the Java StarSs runtime has been adapted to function in Cloud environments, integrating it in a service-oriented and virtualised platform.

On the one hand, the Cloud runtime can orchestrate the execution of **multiple composite applications simultaneously**, each generating its own graph of tasks. These applications can then be part of a service class, offered as operations of a service interface and published as a service in a service container, for clients to invoke them.

On the other hand, the runtime is also able to **interact with virtualised Cloud providers** in order to elastically acquire and release virtual machines depending on the task load that it is processing at every moment. In the same execution, the runtime can dialogue with more than one Cloud provider, which

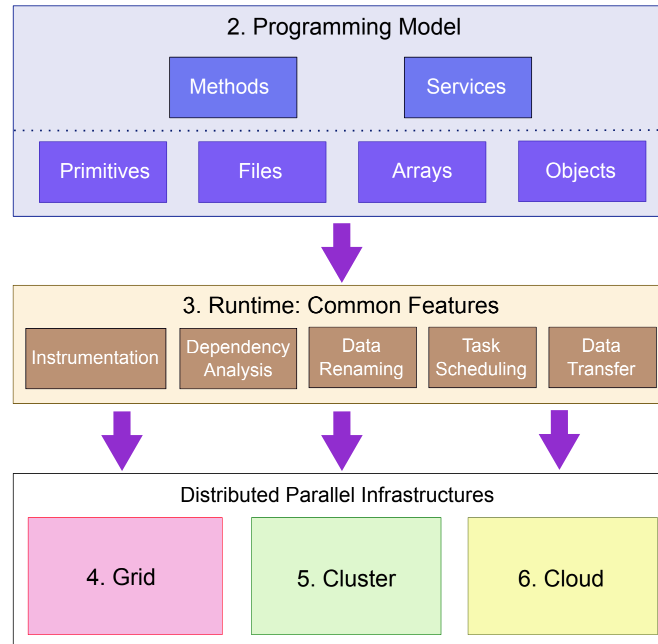


Figure 1.2: Thesis organisation.

facilitates interoperability and makes possible hybrid setups, like the combination of a private cloud with bursting to a public cloud to face peaks in load.

Similarly to the other two scenarios, a Java StarSs application does not contain any Cloud specifics; service orchestration and virtualised resource management are automatically taken care of by the runtime. The work for clouds has been published in [170] and submitted for publication in [136].

1.3 Thesis Organisation

Figure 1.2 depicts the organisation of this thesis.

The two chapters after this introduction present the basic features of the Java StarSs programming model and the runtime system that enables it. More precisely, Chapter 2 provides a comprehensive description of the whole programming model, including the supported types of task (method, service) and data (primitive, file, array, object). Then, Chapter 3 describes the core functionalities of the runtime system, such as dependency analysis or task scheduling, which exist in all its implementations.

After that, three chapters examine each of the distributed parallel infrastructures where Java StarSs has been implemented, namely Chapter 4 for Grid, Chapter 5 for Cluster and Chapter 6 for Cloud. The order of these three chapters corresponds to the chronological order in which the student worked on each

infrastructure, for the reader to understand the challenges found in each case and how the thesis evolved to address them. In fact, both the programming model and the runtime have been modified incrementally, resulting in an implementation that can execute in all these three kinds of infrastructure.

The Grid, Cluster and Cloud chapters follow the same structure, basically divided in three parts: first, an introduction to the context of the infrastructure and the design/technology decisions that it motivated; second, an evaluation of the model on that infrastructure in terms of productivity (i.e. programmability and performance); third, a state-of-the-art section that compares Java StarSs to other models and runtime systems for that particular infrastructure. Therefore, the state of the art is not discussed as a whole at the beginning of the document, but separately at the end of chapters 4, 5 and 6, so that the implementation of Java StarSs for each infrastructure is explained before the differences with other approaches are highlighted.

Finally, Chapter 7 discusses the conclusions of the thesis and proposes some future work. In addition, as a complement of the preceding chapters, Appendix A shows the code and dependency graph of some applications developed and executed with Java StarSs.

Chapter 2

Programming Model

Parallel programming is generally considered to be harder than sequential programming, partly because of the complexity of reasoning, developing, testing and debugging an application in the context of concurrency. Programmers with experience in writing sequential programs usually find it difficult to move to a parallel environment, where they are faced with duties like work partitioning, data partitioning, parallel data access control, synchronisation, communication, etc. Such duties can affect programming expressiveness and make users reluctant to adopt a given parallel language or model.

In that sense, this chapter presents a programming model, Java StarSs, that intends to maximise programmability of Java applications running on parallel and distributed infrastructures. Although the users of this model need to think about opportunities for parallelism when designing their applications, the programming is fully sequential, thus eliminating most of the aforementioned drawbacks of concurrent/distributed programming.

The aim of the model is for the user to code as she would do with a sequential Java application, where built-in control flow statements and primitive types are used; where objects are created, receive method invocations or field accesses, are passed as parameters or returned by methods; where arrays are accessed by referencing their elements; where files are created or opened and read or written by means of streams. Any data type of the Java language can be used, independently of the infrastructure where the application will run.

The next sections gather all the features of the programming model, showing how it can be used to easily parallelise a sequential program. The description of the model is abstracted both from the runtime underneath - introduced in Chapter 3 - and from the particularities of each infrastructure on which it has been implemented - Chapters 4, 5 and 6. The chapter is organised in a first section with a general overview of the model, followed by two sections with a comprehensive specification of its syntax and semantics and how the user should proceed.

2.1 Overview

The central concept in Java StarSs is that of a *task*, which represents the model's unit of parallelism. A task is a method or a service called from the application code that is intended to be spawned asynchronously and possibly run in parallel with other tasks on a set of resources, instead of locally and sequentially. In the model, the user is mainly responsible for identifying and selecting which methods and/or services she wants to be tasks.

A strong point of the model is that *the application is programmed in a totally sequential fashion*; no threading or remote method invocation interface needs to be used. However, at execution time, concurrency is automatically achieved by intercepting the calls to the selected tasks and asynchronously launching them, leaving to a runtime system - explained in Chapter 3 - all the burden of managing the tasks, controlling the data they access and mapping them to the available resources. Such runtime is also in charge of abstracting the application from the infrastructure-related details, so that aspects like resource management or deployment do not appear in the application code.

2.1.1 Basic Steps

The Java StarSs programming model mainly involves thinking about and choosing the right tasks for our application. In order to do that, the user should proceed in two basic steps: identifying the tasks and selecting them. These steps are summarised in Figure 2.1 and discussed next.

2.1.1.1 Identifying the Potential Tasks

In a first step, the programmer determines which will be the tasks of the application. Tasks are entities enclosing a certain computation and they can be of two types:

- Regular Java *methods*.
- *Services*, abstractly understood as a piece of software accessible over a network through a well-defined interface.

Therefore, the current implementation of the model requires the task code to be encapsulated either in a method or a service. In some cases, the application may already be formed by calls to computationally-intensive methods or services that are clear candidates to become tasks. Nevertheless, sometimes it may be necessary to do some code refactoring in order to delimit what will be a task, especially when the programmer does not start from scratch but from an already existing sequential application. Figure 2.2 depicts an example of such a situation: the code in (a) increments every element of a matrix A. Let us assume that the user wants the inner loop that increments a row of the matrix to be a task. For that purpose, the loop is encapsulated in method incrementRow, as shown in (b).

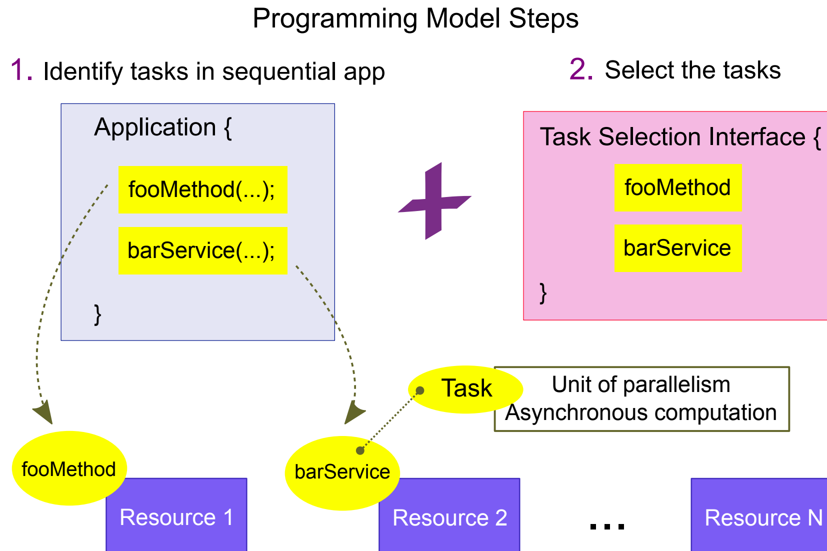


Figure 2.1: Steps of the Java StarSs programming model. In the application, which is programmed sequentially, the user identifies the methods and services to be tasks and then selects them. The model is based on inter-task parallelism and task asynchrony.

Another aspect that the programmer should take into consideration is task granularity. There is a general tradeoff when choosing the granularity of tasks: more and smaller tasks help achieve better load balance when the application runs whereas, on the contrary, fewer and coarser-grained tasks incur in less task management and communication overhead. For example, in Figure 2.2(b), the granularity of incrementRow (the amount of computation) is controlled by the number of columns of the matrix (NCOLS). On the other hand, depending on the infrastructure, the adequate granularity can vary (e.g. coarse grained in grids, finer in clusters); however, this granularity can be a parameter of the application, so that the code of the latter does not have to be adapted to each infrastructure.

2.1.1.2 Defining a Task Selection Interface

Once the user has figured out which will be the application tasks, the next step is selecting them. In order to do that, the user defines a Java interface which declares those methods and services to be the application tasks.

The task selection interface is not a part of the application: it is **completely separated from the application code** and it is not implemented by any of the user's classes; its purpose is merely specifying the tasks.

All the information needed for parallelisation is contained in this separate interface, and not in the application code. In particular, each method or service

```
// A is a matrix of NROWSxNCOLS integers
for (int i = 0; i < NROWS; i++)
    for (int j = 0; j < NCOLS; j++)
        A[i][j]++;
```

(a)

```
for (int i = 0; i < NROWS; i++)
    incrementRow(A[i]);

...

public static void incrementRow(int[] row) {
    for (int j = 0; j < NCOLS; j++)
        row[j]++;
}
```

(b)

Figure 2.2: Example of code refactoring. An application that increments the rows of a matrix (a), composed by two loops, is reorganised to encapsulate the inner loop in a method incrementRow (b) so that it can become a task.

declared in the interface must be accompanied by some metadata to uniquely specify it and to state how its parameters are accessed. More details about these metadata and the way they are provided will be given in Section 2.2.

2.1.2 Sequential Programming

The applications that follow the Java StarSs model are programmed in pure sequential Java. The user is *not required to include any API call* or special pragma in the code. Moreover, the invocation of a task (for both methods and services) is performed on a regular Java method, and the application data is also accessed as normal.

Even though tasks are asynchronously submitted to remote resources, the model ensures *sequential consistency* of the parallel execution. In other words, the results of the application are guaranteed to be the same as if it ran serially on a single core. Actually, the user can test the application by running it sequentially and also debug it locally; once the program behaves as desired, it can be parallelised with the model, thus simplifying the testing and debugging stages of the application development.

Regarding the application data, on the one hand, the user can select tasks that share data through their parameters and, on the other, those data can also be read and/or written later from the sequential part of the application; in neither of those cases the user is aware of data being transferred back and forth, data versioning or data access synchronisation. Moreover, the user does not control the data layout of the application in a distributed execution. All of this is taken care of transparently by the runtime, which makes sure that the application performs its accesses on the right data versions and which manages data locations, as will be seen in Chapter 3.

The way the programmer utilises the diverse features of the Java language and its implications at Java StarSs level will be examined in Section 2.3.

2.2 The Task Selection Interface

The task selection interface is the means to tell Java StarSs about the tasks. Each entry of the interface selects and describes a task method or service, and is composed by two parts: first, the declaration of the method/service itself, formed by its name, formal parameters and return value; second, some metadata about the task, specified as Java annotations [31], which are a subset of the Java syntax. Annotations consist of an at-sign (@) followed by an annotation type and a list of element-value pairs in parentheses. These annotations are used by the programmer to provide task information both at method level - "what method/service am I referring to?" - and at parameter level - "how does the task access its parameters?".

Consequently, a Java StarSs programmer is responsible for choosing and describing the tasks. Alternatively, Java StarSs could transform every single method invocation into a task, but some of these methods could be too fine-grained for them to be worth distributing. In that sense, the programmer knows her application and can better decide which subset of methods are suitable to become tasks. On the other hand, we are investigating techniques to automatically infer how task parameters are accessed, but such research is out of the scope of this thesis.

Figure 2.3 defines the syntax of the task selection interface, which will be explained in the next subsections.

2.2.1 Method-level Annotations

Every task method or service in the interface must be preceded by an annotation that marks it as such and describes it.

2.2.1.1 @Method

The `@Method` annotation is associated to a method task. It contains the following elements:

- *declaringClass* (mandatory): fully-qualified name of the class that contains the implementation of the method. It allows to uniquely identify a method, together with the name of the method and the type of its parameters, which can be extracted from the method declaration itself.

- *isModifier* (optional, default `true`): when set to `false` for an instance method (i.e. non-static), it indicates that the method only reads the object it is invoked on. The use of this element will be exemplified in Section 2.3.4.

- *isInit* (optional, default `false`): when set to `true`, the task is marked as an *initialisation task*. Even though the programmer cannot specify the data

```
public interface class_nameItf {

    [@Constraints(property_name = "property_value")]
    @Method(declaringClass = "package_name.class_name"
            [, isInit = [true | false]] [, isModifier = [true | false]])
    return_type method_name(
        [@Parameter([direction = [IN | OUT | INOUT]] [, type = FILE])]
        parameter_type parameter_name
    );

    @Service(namespace = "service_namespace", name = "service_name",
            port = "service_port")
    return_type operation_name(
        parameter_type parameter_name
    );

}
```

Figure 2.3: Syntax of a task selection interface, comprised of a method task and a service task declarations. The annotations are represented in bold: @Method for identifying a method, @Service for a service operation, @Constraints to specify the resource requirements of a method task and @Parameter to state the direction and type of a method task parameter. The elements of each annotation are in italics.

layout of the application, she can use initialisation tasks to distribute data uniformly among the available resources. Usually, they are methods that allocate (and return) data in the resource where they run, and they are especially treated in terms of scheduling. The usage and behaviour of this kind of tasks will be further discussed in Chapter 3, Sections 3.5 and 3.6.

For a method to be a task, it must only fulfill a couple of restrictions. First, the method must be pure, that is, it cannot access global data, only its parameters and local variables. Second, the parameters, return value and callee object (if any) of the method must be serializable - i.e. implement the standard Java Serializable interface - for them to be sent over a network.

Java StarSs tasks can be either an instance method or a class method. Furthermore, they can either be void or return an object, an array or a primitive type.

2.2.1.2 @Service

The @Service annotation is associated to a service task, which corresponds to a Web Service operation. A Web Service [68] is commonly defined as a software system that supports interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (WSDL [69]), containing the operations it offers. Other systems interact with a Web Service in a manner prescribed by its description using SOAP [55] messages, typically conveyed using HTTP [25].

In the task selection interface, service tasks are declared as normal Java methods whose name and parameters match exactly those of the service operation to which they refer. Besides, this declaration comes along with a @Service annotation, which has the following elements that complete the identification of the service operation:

- *namespace* (mandatory): namespace of the service, i.e. the context for the identifiers of the service.
- *name* (mandatory): name of the service.
- *port* (mandatory): service port where the operation is defined.

2.2.1.3 @Constraints

Optionally, the programmer can utilise a third type of method-level annotation, only for method tasks. The @Constraints annotation allows to specify the set of capabilities that a resource must have in order to execute the task. Thus, the user can demand, for instance, some processor-related characteristics (architecture, number of CPUs, GHz), memory, storage capacity or operating system.

Please note how these constraints do not tie the application to a particular infrastructure because they are not a part of the application code, instead they are placed in the task selection interface. Moreover, they are not mandatory, they can be optionally used by the programmer to make sure that some resource requirements of the tasks are fulfilled when scheduling them.

2.2.2 Parameter-level Annotations

Method tasks need an additional annotation at parameter level. The main purpose of this annotation is to state how the task accesses its parameters, i.e. in read, write or read/write mode. This information is of utmost importance because it permits to control dependencies on data accessed by several tasks. How the Java StarSs runtime detects data dependencies based on this information will be explained in Chapter 3, Section 3.3.

Service tasks, on the other hand, do not require the programmer to specify any parameter access mode. The parameters of a web service operation are always read-only: they are sent to the server running the service and whatever happens in the operation code remains hidden to the user; the response of the operation is provided through its return value.

2.2.2.1 @Parameter

The @Parameter annotation precedes each parameter of a method task. It can have two elements:

- *direction* (mandatory in some cases, default IN): direction of the parameter, i.e. how the parameter is accessed inside the task. It can be IN (read mode), OUT (write mode) or INOUT (read/write). It is not necessary

for primitive types, for which direction is assumed to be IN because Java primitives are always passed by value: the actual parameter is copied into a location that holds the formal parameter's value.

- *type* (only mandatory for files): Java StarSs type of the parameter. A task parameter can be of any type supported in Java: primitives, objects and N-dimensional arrays. In addition, Java StarSs features a special type FILE intended for method tasks that work with files; in that case, what the user passes as parameter is a String object containing the path to the file. In most occasions, the user does not need to specify the type of the parameter, since it can be automatically inferred from its formal type in the method declaration; in particular, this can be done with objects, arrays and primitives. However, for file parameters, the pair "type = Type.FILE" must appear in the annotation; in this case, the Java StarSs type (FILE) cannot be deduced from the formal type (String): the user needs to clarify whether that string is a file URI or not.

Note that, for objects or arrays with IN direction and primitive types, none of the elements listed above is mandatory, and therefore the @Parameter annotation can be omitted.

2.3 The Main Program

Once the user has finished the task selection, the application is conceptually divided in two parts (see Figure 2.4):

- *The tasks*: code that runs asynchronously in a certain resource.
- *The main program*: non-task code. It is completely sequential and executes on the resource where the application is launched.

The main program is the part of the application where the programming model features are applied. In other words, it is the code from where the user can invoke tasks and eventually access their data, always writing in sequential Java. In order to keep the simplicity of the model and, at the same time, make possible the parallelisation, there is a need for a runtime system that steers the application execution.

In practice, such steering is not performed in every single class that is referenced from the main program, but on a restricted set of classes that is configurable by the user. Consequently, the main program becomes divided in two areas: one that is under supervision and a *black box* whose code runs serially with no intervention of the runtime. The reasons of this division will be explained in Chapter 3, which will also thoroughly describe the runtime support for all the programming model features that are presented in this chapter.

The rest of this section first overviews two possible scenarios where the programming model can be applied, which differ in how the main program is exposed for execution. After that, it discusses how, in the context of the main

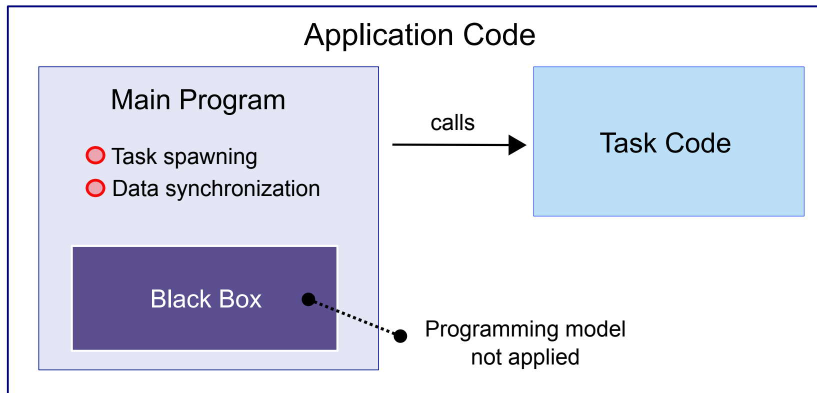


Figure 2.4: Parts of the application code: main program and task code. In the main program, except for the black-box area, the programming model features are enabled.

program, the user deals with the key aspects of the model: invoking a task, passing data to a task and working with data created/modified by a task.

2.3.1 Scenarios

The Java StarSs programming model contemplates two types of application, listed next.

2.3.1.1 Regular Application

The first kind (Figure 2.5(a)) is a regular sequential Java application, formed by the application class, which contains the `main` method, and the rest of classes that are directly or indirectly referenced from that class. The `main` method is the entry point for the main program and the first code that runs when launching the application.

2.3.1.2 Composite Service

The second kind (Figure 2.5(b)) is an application accessible as a service. In this case, the main program is actually a service operation (`compositeService` method) implemented in a service class and offered in a service interface to external users. Hence, the entry point for the main program is a web service invocation which leads to the execution of the corresponding method. An `@Orchestration` annotation must accompany a service operation method where the programming model is applied.

This scenario allows to create *composite services* as sequential Java programs from which other services and regular methods are called. Therefore, composites can be hybrid codes that reuse functionalities wrapped in services or methods,

```
public class RegularAppClass {

    public static void main(String args[]) {

        ...     // main body

    }

    ...     // rest of the class

}
```

(a)

```
public class ServiceClass {

    @Orchestration
    public static Response compositeService(...) {

        ...     // main body

    }

    ...     // rest of the class

}
```

(b)

Figure 2.5: Examples of two scenarios for a Java StarSs application: (a) regular application with a main method that starts the execution, (b) composite service operation provided by a remotely-accessible web service.

adding some value to create a new product that can also be published as a service. This kind of applications fit in the area of Cloud Computing, which will be addressed in Chapter 6.

2.3.2 Invoking Tasks

In Java StarSs, *tasks are always invoked like a normal Java method*, no matter whether they correspond to methods or to service operations. Figure 2.6 provides examples of invocations for both kinds of task.

2.3.2.1 Methods

As introduced earlier in this chapter, method tasks correspond to Java methods implemented in a certain class. Even if a given method has been selected as a task, its invocation from the main program remains the same.

An example can be found in Figure 2.6(a), line 1, where a task method sampleMethod is called. Here, the method is static (although instance methods are also supported, see Section 2.3.4) and returns an object of type Value. When the application reaches line 1, a task for sampleMethod is asynchronously spawned, thus letting the program continue its execution right away.

```
    // method task invocation
1   Value val = sampleMethod();

    // stateless service task invocation
2   statelessServiceOp(val);

    // stateful service task invocation
3   SampleService s = new SampleService();
4   s.statefulServiceOp(val);
```

(a)

```
public interface SampleItf {

    @Method(declaringClass = "sample.Sample")
    Value sampleMethod();

    @Service(namespace = "http://sample.com/sample",
             name = "SampleService", port = "SamplePort")
    void statelessServiceOp(
        Value val
    );

    @Service(namespace = "http://sample.com/sample",
             name = "SampleService", port = "SamplePort")
    void statefulServiceOp(
        Value val
    );

}
```

(b)

Figure 2.6: Examples of task invocations from a main program (a), for both methods and services (including stateless and stateful-like invocations). In (b), the corresponding task selection interface is shown as a complement to the main program in (a).

2.3.2.2 Services

The programming model also allows to execute service operations as tasks: the invocation of a service task from the main program leads to the creation of an asynchronous computation, like for any other task.

Services are external software entities accessible through a network and, as such, they require a mechanism for the user to invoke them from the application. In this sense, the invocation of service tasks is not different from that of method tasks: service operations are called as regular Java methods as well. Chapter 3, Section 3.2.4.1 will explain how the mechanism for service task invocation is exactly implemented.

On the other hand, a service can have an internal state that might be modified when running one of its operations. In that regard, the model offers two ways of calling a service operation: *stateless*, if the operation does not change the state of the service, and *stateful*, if it does.

An example of a stateless service invocation is given in Figure 2.6(a), line 2. The statelessServiceOp operation is called by means of a static representative method with its same name and parameters.

A stateful service invocation is slightly different: first, in line 3, an object of class SampleService - which can be seen as the state of the service - is created; then, in line 4, the operation is invoked on that object. Therefore, this time the call is performed on a class representative method declared in class SampleService, named after the service. Stateful invocations may modify the internal state of the service and, consequently, Java StarSs guarantees that they will be serialised so that the state is updated in mutual exclusion.

2.3.3 Sharing Data Between Tasks

Tasks are not isolated, they can access data coming from both the main program and other tasks. The way for tasks - both methods and services - to share data is by means of their parameters, return values and callees, never through global data of the application (e.g. static class fields). Data created in the main program can later be handled by a task, and also data produced or accessed by a task can be reused by a subsequent task. Nevertheless, the user does not have to explicitly control the various data versions nor the possible data dependencies between tasks or between a task and the main program; this is all managed by the runtime, as will be seen in Chapter 3.

As an example of task data sharing, in Figure 2.6(a), the val object returned by sampleMethod in line 1 is then received as a parameter by sampleServiceOp in lines 2 and 4.

2.3.4 Working with Objects

Like in other object-oriented languages, objects are the main concept which drives the developing of a Java application. Hence, any Java-based parallel programming model must address the issue of combining objects and concurrency.

In Java StarSs, objects can be created and used in the main program of the application, and they can also appear anywhere in the signature of a task. The aim is for the programmer to code as she would do with any sequential application where objects are created, receive invocations or field accesses, are passed as parameters or returned by methods.

The next subsections will go through the different aspects of programming with objects in the model. A sample application, depicted in Figure 2.7, will illustrate some of the explanations. Although this example uses a method task, it could also have been done with a service task.

2.3.4.1 Objects in a Task

Figure 2.7(a) shows how Java objects created in the main program of the application (a and b, lines 1-2) can eventually be used by a task afterwards (line 3, foo method, selected in the interface in Figure 2.7(b)).

```
1  A a = new A();
2  B b = new B();
3  C c = a.foo(b);      // call to a selected (task) method
   ...                  // other statements
4  c.bar();             // synchronisation by method call on c
5  int i = a.f;         // synchronisation by field access on a
```
(a)

```
public interface AppItf {
    @Method(declaringClass = "example.A")
    C foo(
        @Parameter(direction = IN)
        B b
    );
}
```
(b)

Figure 2.7: Sample sequential application (a) and its corresponding task selection interface (b). Method foo, implemented by class example.A, is chosen as a task in (b); it is an instance method (invoked on an object of class A), receives an input object parameter of class B and returns a C object.

Just like in any regular method call, a task object can be located in three different positions, discussed next.

*** Callee**

Object a is the callee of method foo, i.e. the target object on which the method is invoked. foo is an instance method implemented in example.A, specified in the interface as its declaring class, so it is invoked on an object of class A. By default, the type of access on the callee object is assumed to be INOUT, but the programmer can change it to IN by adding an isModifier element with value false to the @Method annotation; this element can be helpful for avoiding a data dependency between foo and a subsequent task that reads its callee (please refer to Chapter 3 for more information about data dependency control):

```
@Method(declaringClass = "example.A", isModifier = false)
C foo(...);     // foo accesses its callee object in read mode
```

In addition to instance methods, static ones are also supported. The declaration in the interface for both kinds is equivalent, but all the considerations just discussed do not apply on the latter since they are not invoked on any object.

*** Parameter**

Object b is passed as parameter of method foo. In the declaration of foo in the interface, the parameter is defined to have IN direction (@Parameter annotation), i.e. the foo task will read it. Notice that the direction could also

be INOUT or OUT if the parameter were read/written or only written by the task, respectively.

*** Return Value**

Object c is the return value of method foo. This case differs from the other two in the fact that the object is not created in the main program but inside a task. Conceptually, a return value is like a parameter that has always OUT direction, since it is a result produced by the task.

In order to keep the asynchrony in the generation of tasks that return an object, the programming model features *future objects* [109]: whenever there is a call to a task method returning an object, the task is immediately spawned and an object of the same class, the future, is created for it to take the place of the not-yet-generated object in the main program. This requires the class of the object to have an empty constructor with no arguments, which will be used to create the substitute object and return it right away.

For the programmer, future objects are just like any other object. Thus, in the main program, the object returned by a task call can then be accessed or passed to another task. Synchronisation is completely transparent to the programmer, as will be explained next in Section 2.3.4.2.

2.3.4.2 Access in Main Program

Any object that participates in a task call can be accessed later on in the main program. From the point of view of the programmer, the use of an object is not different whether it has been created/accessed by a task before or not.

However, in order to guarantee the sequential memory consistency of the application, when the main program accesses an object previously produced or updated by a task, a synchronisation is needed to fetch the right (last) version of the object. The programmer is completely unaware of such process and codes as if the application had to be run sequentially.

The next points describe the different kinds of access to an object that can be detected and synchronised. More details about how this is done will be given in Chapter 3, Section 3.2.

*** Method Call**

In Figure 2.7(a), object c, which was returned by the task method foo in line 3, is next accessed from the main program in line 4 by invoking the bar method on it. The bar call on c is done as any other method call.

It is worth pointing out that the synchronisation for c is delayed until line 4, when bar is invoked and the last value for c is truly needed. This fact helps increase parallelism, since other statements between line 3 and 4 (possibly including task invocations) can execute before the main program is blocked to synchronise.

*** Field Access**

An object can also have one of its fields accessed. Figure 2.7(a) shows an access on field f of object a in line 5. A synchronisation is triggered at that point because the foo task invoked in line 3 generates a new version of a (callee

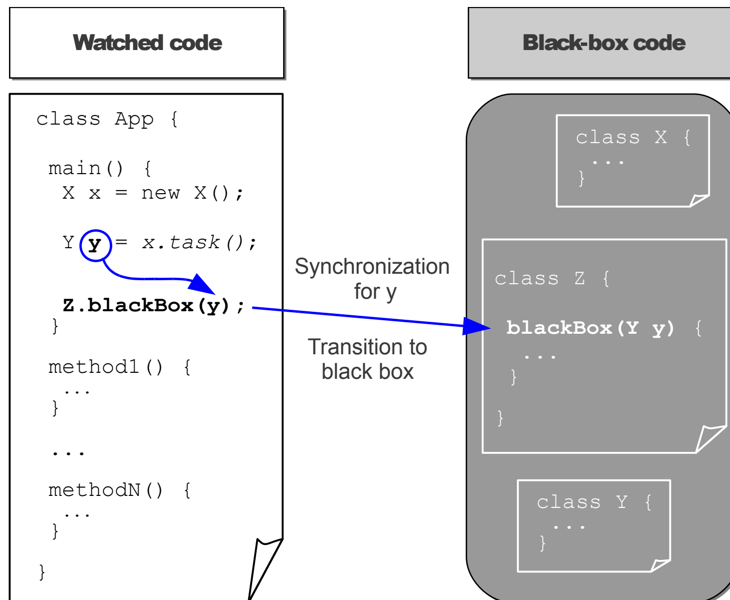


Figure 2.8: Case of synchronisation by transition to black-box area. Object y is returned by method task of class X, which we assume was selected as a task and therefore is spawned asynchronously. When the main program reaches the call to method blackBox, which is implemented in a non-watched class Z, a synchronisation takes place to get y and pass it to blackBox.

objects have INOUT access mode by default), possibly modifying field f. Hence, the last value for f has to be obtained and stored in i.

*** Transition to Black-Box Area**

The accesses to fields and the invocation of methods on objects are not watched in those classes that are part of the black box of the main program (see Section 2.3); therefore, the corresponding synchronisation mechanisms do not take place when running code inside those classes.

However, as illustrated in Figure 2.8, there is a kind of synchronisation by transition from watched code to the black box. This happens when, from supervised code, there is a call to a method or a constructor of a class which is part of the black box. If such method/constructor call receives as parameter any object previously accessed by a task, a synchronisation is started. Such action is necessary because the black-box code may read the object and therefore the right value has to be passed.

2.3.5 Working with Arrays

Arrays in Java are objects, but a special kind of object. They act as containers of a fixed number of elements of the same type. The elements of an array are not

```
1  int[] iArray = new int[SIZE];
2  B[][] bArray = foo(iArray);      // foo is a task
   ...                              // other statements
3  int i = iArray[0];               // synchronisation by element access (1D)
4  B b = bArray[1][2];              // synchronisation by element access (2D)
```

Figure 2.9: Example of synchronisation by access to an array element from the main program. foo is assumed to be a task that receives a one-dimensional array as a parameter, updates it and returns a two-dimensional array. Those arrays are accessed later in the main program, each forcing a synchronisation.

accessed with variable names, instead they are referenced by array expressions that use non-negative integer index values. Moreover, the elements of a Java array can be references to other arrays, thus forming multidimensional arrays.

2.3.5.1 Arrays in a Task

The positions of an array in a task are analogous to those of the rest of objects, namely callee, parameter and return value. Nevertheless, an array is unlikely to be the callee of a task, since the only methods that can be invoked on an array are those inherited by the Object class, i.e. those that are common to all objects, which provide very basic functionalities.

2.3.5.2 Access in Main Program

In addition to the kinds of access that apply to objects (method call, field access and transition to black box), arrays are most typically accessed by referencing one of their elements.

*** Access to an Element**

Figure 2.9 depicts an example of synchronisation by access to an array element. Line 1 allocates a 1D array of integers which is later passed as a parameter of method foo in line 2. This method, which is assumed to be selected as a task, both reads and writes its parameter and returns a 2D array of B objects.

Line 3 reads the value of the element in position 0 of iArray, which causes a synchronisation to get that array, modified by foo. Similarly, in line 4, the synchronisation ensures that the 2D access on bArray will be done on the array returned by foo. The same mechanism would be applied for any N-dimensional array.

2.3.6 Working with Primitive Types

The Java programming language supports eight pre-defined primitive types, namely byte, short, int, long, float, double, boolean and char. Character strings are objects of the String class and therefore they are not a primitive type.

2.3.6.1 Primitives in a Task

Primitive types can be either a parameter of a method or a return value, and so these are the two positions where they can be found in a task.

When passed as parameters, primitives have always IN direction, because in Java they can only be passed by value. On the other hand, like for objects and arrays, when a primitive type is returned by a task its direction is implicitly OUT.

2.3.6.2 Access in Main Program

Since primitive types that act as parameters of a task are always passed by value, this eliminates the need for synchronising them later in the main program. In Figure 2.10, boolean b is an actual parameter of foo, a task method, called in line 2; however, no synchronisation takes place when reading b in line 3.

Concerning primitives that are returned by a task, they are not objects and consequently they cannot be replaced by futures. The synchronisation, in this case, is immediate. In Figure 2.10, integer i is returned by foo in line 2; since i might be required later, foo cannot be asynchronous and the main program must wait for the task to generate i and then get that value. Hence, when reaching line 4, the main program has already got the value and can run the increment statement safely.

```
1  boolean b = false;
2  int i = foo(b);        // foo is a task, immediate synchronization for i
   ...                    // other statements
3  boolean c = b;         // no need to synchronise
4  i++;                   // previously synchronised
```

Figure 2.10: Example of synchronisation for primitive types. The invocation of the task method foo is synchronous here, because of the integer value i that it returns. Primitive types that are passed as task parameters, like b, do not require synchronisation.

2.3.7 Working with Files

In addition to data in memory, a Java programmer also has means to work with files. Arguably, one of the first things a programmer learns with a new language is how to read and write to files, since the saving and loading of data is an important feature of most software.

Java offers a rich variety of file handling classes, which are mainly based on the use of *streams*: input streams are sources of data, whereas output streams are destinations for data. Besides, I/O operations can deal either with raw bytes or with characters in a given encoding.

```
1  String myFile = "/path/to/myfile/counter";

2  FileOutputStream fos = new FileOutputStream(myFile);
3  fos.write(VAL);
4  fos.close();

5  increment(myFile);      // call to a task method

   ...

   // synchronisation by input stream opening
6  FileInputStream fis = new FileInputStream(myFile);
7  int i = fis.read();
8  fis.close();

9  increment(myFile);      // call to a task method

   ...

   // no need to synchronise: out access
10 FileOutputStream fos = new FileOutputStream(myFile);
11 fos.write(NEW_VAL);
12 fos.close();
```

(a)

```
public interface AppItf {
    @Method(declaringClass = "example.myClass")
    void increment(
        @Parameter(direction = INOUT, type = FILE)
        String file
    );
}
```

(b)

Figure 2.11: Sample sequential application (a) and its task selection interface (b). Method increment is chosen as a task in (b); it receives an input/output file parameter where a counter value is stored and increments that value. In (a), the main program opens streams on the same file incremented by the task.

The next subsections will go through the different aspects of programming with files in the model. A sample application, depicted in Figure 2.11, will illustrate some of the explanations.

2.3.7.1 Files in a Task

Files can be parameters of a method task. The way to pass a file to a task is by means of a String object that contains the path to the file. In Figure 2.11(a), a String that refers to a sample file is created in line 1; then, in line 2, an output stream is opened on the file using that string, in order to write an initial value for the counter (line 3); line 4 closes the stream; finally, a task method increment is invoked passing the string as a parameter.

Regarding the task selection interface (Figure 2.11(b)) files are the only case

in which the programmer must explicitly specify the data type of the parameter; because the real type of the parameter is String, it is necessary to state that such string represents a file path. On the other hand, since method increment reads the counter value in a file, increments it and writes it back to that file, the direction of the parameter is INOUT.

2.3.7.2 Access in Main Program

The programming model permits to open streams on files in the main program, even if those files are written by a previously-spawned task. It is guaranteed that, when necessary, the right version of the file will be obtained before the stream is created on it. Again, such process is transparent to the programmer, who works with the stream in a normal way.

In Figure 2.11(a), line 6, an input stream is opened on myFile in order to read its counter value (line 7). This triggers a synchronisation to get the value incremented by task increment, invoked in line 5.

A second call to increment happens in line 9, thus generating a second asynchronous task. When the program reaches line 10, another stream is opened on myFile; however, since this time it is an output stream that truncates the file, there is no need to wait for the value of the second increment: the stream is created immediately and a new value is written in the file (line 11).

Besides the ones in Figure 2.11(a), the model also supports several other kinds of streams and file handling classes, including buffered streams (like BufferedInputStream) and character streams (like FileWriter).

2.4 Summary

This chapter has provided a whole view of the Java StarSs programming model and constitutes a specification of its syntax, semantics and usage.

The main purpose of the model is to hide the complexity of developing Java applications for parallel and distributed infrastructures. Writing such an application with this model only requires sequential programming skills: the application code is in plain serial Java, with no changes to the way the programmer invokes methods and works with data, and no need to include any library call, pragma or infrastructure-related detail.

In Java StarSs, the user is primarily responsible for identifying and choosing the application tasks, which are the model's unit of parallelism: concurrency is achieved by asynchronously spawning tasks to a set of available resources. Tasks are methods or services called from the application and selected by the user in a separate Java interface, which contains all the information needed for parallelisation. Even if the programming is sequential, it is advisable that the user think about opportunities for concurrency and task granularity when choosing the tasks.

So far, little information has been given about how the application is actually run, i.e. how the various features of the programming model are made

possible at execution time. In that regard, the model is supported by a runtime system that is in charge of managing the execution of the application; its general characteristics will be described in Chapter 3. Furthermore, some of the aspects of the model were conceived for or fit better in a particular environment; hence, the three infrastructure chapters (4, 5 and 6) will focus on different parts of the model. Those chapters will also complement the current chapter with examples of real applications and benchmarks programmed with Java StarSs.

Chapter 3

Runtime: Common Features

As a result of following the steps of the Java StarSs programming model, presented in Chapter 2, the user ends up with two outcomes: (i) a sequential application and (ii) an interface that selects the tasks. Clearly, these two elements by themselves do not enable parallelisation: there is a need for a runtime system that, taking (i) and (ii) as input, provides the magic and brings the model's features into action.

In that regard, this chapter describes the core functionalities of the runtime system on top of which the programming model is built. This includes generating asynchronous tasks, watching data accesses from the main program, controlling task dependencies, transferring data, scheduling, submitting and monitoring tasks; all this is done automatically and transparently to the programmer, and keeping the application agnostic to the infrastructure. Moreover, the chapter is a link between the programming model characteristics - what the user writes - and their implementation - what happens at execution time.

The functionalities discussed here are common to the three distributed infrastructures considered in this dissertation (Grid, Cluster and Cloud). The next chapters (4, 5 and 6) will show how the design of the runtime was adapted to each infrastructure to address its particularities.

The chapter is organised in a first section about the general structure of the runtime, followed by six sections, each corresponding to one of the core functionalities. The functionalities are presented as they manifest during the application execution, so that the reader can get a better picture of the entire execution process.

3.1 General Structure

The Java StarSs runtime is organised in a master-worker structure, as depicted in Figure 3.1:

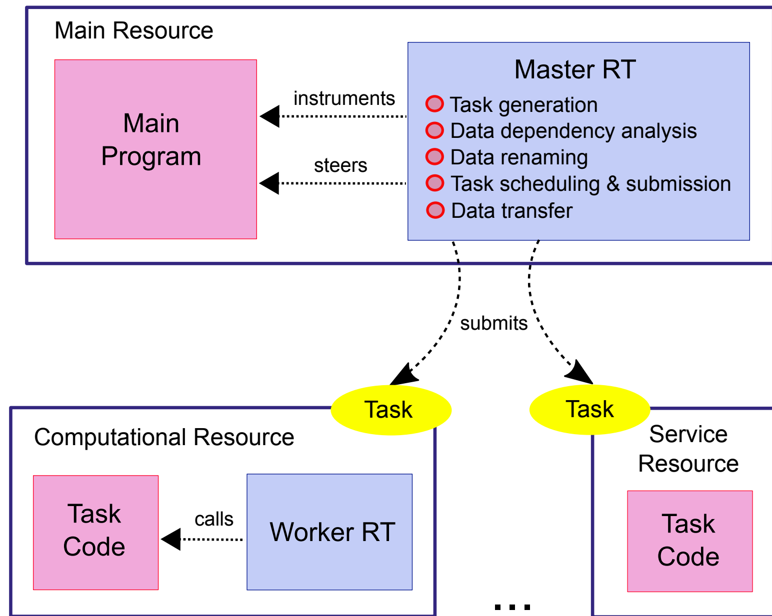


Figure 3.1: Java StarSs runtime structure. The master side deals with the main program of the application, whereas the worker side handles the task code.

- *Master*: the master part executes in the resource where the application is launched, i.e. where the main program runs. It can be described as the 'brains' of the runtime: it is responsible for steering the parallelisation of the application and implements most of the features of the runtime, which basically concern task processing and data management. In a first phase, the master runtime inserts some code in the application to spawn tasks and synchronise data; when the main program starts running, tasks are asynchronously generated and the runtime inspects which data they access and how, thus discovering the data dependencies between them; after that, the dependency-free tasks are scheduled on the available resources; finally, the master runtime transfers the input data of the tasks to their destination resources, submits them for execution and controls their completion.

- *Worker*: the worker side of the runtime is mainly in charge of responding to task execution requests coming from the master, although in some designs of the runtime it also has data transfer capabilities, as will be seen in Chapter 5. On the other hand, the worker runtime is only present in worker computational resources, which typically correspond to a node of a cluster or grid, or a virtual machine in a cloud; in service resources, which are services deployed in an external container, the task code is executed with no intervention of the runtime on the server side.

3.2 Bytecode Instrumentation

As introduced in Chapter 2, Section 2.3, the main program of the application is divided in two parts: one that is under supervision of the Java StarSs runtime, where the programming model features are applied, and another one that is seen as a black box, whose code runs normally. The first part is the one *instrumented* by the runtime. In short, instrumenting the main program means inserting some logic in it to:

- Replace the calls to the selected methods and services by the asynchronous creation of their associated tasks.
- Watch the data accesses, in order to ensure the sequential memory consistency of the execution.

Figure 3.2 shows the elements that intervene in the instrumentation process and how they interact.

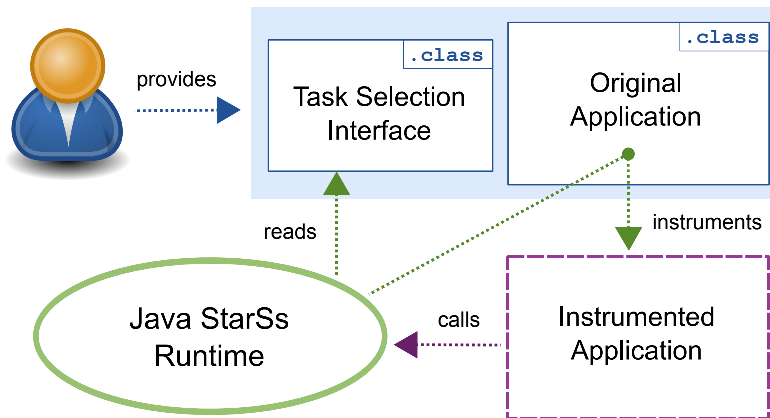


Figure 3.2: As a result of applying the programming model, the user provides the Java classes corresponding to the task selection interface and the sequential application. In order to enable its parallelisation, the application is instrumented to insert calls to the Java StarSs runtime at certain key points. At execution time, the runtime will use the information in the interface to parallelise the instrumented application.

3.2.1 How?

The runtime instruments the application with the help of Javassist [33], a Java library for class editing. This tool can be used for Aspect-Oriented Programming [129], which allows to express cross-cutting concerns of a program in stand-alone modules called 'aspects'. An aspect is essentially the combination of some code - the 'advice' - plus the point of execution in the application where that

code needs to be applied - the 'pointcut'. Javassist works with the bytecode of the application, provided by the user; bytecode is the format to which Java is compiled and that the JVM knows how to interpret, i.e. the classes.

In Java StarSs, the pointcuts are both the calls to the selected tasks and the accesses to their data from the main program, while the advices are the code of the runtime that handles those events. In other words, the instrumentation mainly permits to intercept and respond to a set of key events, thanks to the insertion of some additional code in the original application that checks: first, whether a given method invocation corresponds to a selected task; second, whether it is necessary to synchronise the data involved in a certain access.

3.2.2 When?

The instrumentation is always performed before the application starts to run; indeed, only an instrumented application, i.e. containing calls to the runtime, can be parallelised. However, depending on the type of application (see Chapter 2, Section 2.3.1), the instrumentation is done on-the-fly or offline.

For regular applications, the instrumentation usually takes place when launching them, in a dynamic way: before loading an application class which does not belong to the black box, the runtime first instruments it and then lets the modified class be loaded into memory.

In the case of composite service applications, class instrumentation happens before the service is published. Hence, classes are instrumented statically, the modified bytecode is stored in class files and then these files are included in the service package; the package is later deployed in a container and the service becomes ready for execution. Note that regular applications could also be instrumented offline, but most users find it simpler to do it in a single step when starting the program.

3.2.3 What?

By default, the runtime only instruments the class containing either the main method (for a regular application) or the composites (for applications deployed as services). The user can tell otherwise by defining an instrumentation path, whose concept is similar to that of the Java class path, but in this case it specifies the classes to be instrumented.

Typically, amongst all the classes referenced by a given application, the user only calls tasks and eventually accesses their data from a certain subset of classes; therefore, it makes sense to instrument only the latter and leave the rest untouched. Examples of classes that are likely to belong to the black box are standard Java libraries (lists, hash tables, etc.) or classes in external packages that were not programmed by the user. The instrumentation does add some overhead to the execution, due to the extra checks inserted in the code, and so it is advisable to restrict its scope; in general, though, such overhead is negligible when working in distributed environments.

3.2.4 What For?

As introduced in Chapter 2, Section 2.1.2, the Java StarSs model frees the user from including any invocation to the runtime in the application; instead, this is done automatically by instrumenting its bytecode. Thus, the purpose of the instrumentation process is to produce a modified application that is able to intercept certain events and inform the runtime about them, so that the latter can take the necessary actions to respond to those events and steer the parallelisation of the application.

The next two subsections discuss the kinds of event that need to be watched by the runtime and the behaviour that they trigger.

3.2.4.1 Asynchronous Task Generation

As seen in Chapter 2, Section 2.3.2, both method and service tasks are invoked from the main program as regular Java methods. In the case of method tasks, the invocation is performed on the same method that was selected as a task, implemented by some Java class. Regarding services, the actual method invoked is a local representative of the service operation with the same signature. Every web service interface specifies the operations it provides and the data types that these operations use; with this information, a representative for the task service operation is automatically generated along with the necessary Java types, for the programmer to use them in the main program.

Both for method and service tasks, the invocation needs to be substituted by the spawning of an asynchronous task. This means that, neither a method corresponding to a method task nor a representative of a service task is executed locally; instead, in both cases the runtime must add a task to the graph and, at some point, execute the method in some resource / call the service operation.

In that sense, the instrumentation phase intercepts every method call that is performed from the main program and checks if the invoked method corresponds to a selected method or service task. In order to do that, the runtime reads the content of the task selection interface, i.e. the declared methods and their attached annotations, and compares them with the called method. The compared information depends on the kind of task:

- A match is found for a method task when the called method has the same signature (name and parameter types) and declaring class as that task.
- A match is found for a service task when the called method - the local representative - has the same signature as the service operation and the package of the method's class is a concatenation of the namespace, service name and port name of the operation.

When a given call is identified as a task invocation, the runtime replaces the original call by the creation of a task containing the information of that method or service operation; at execution time, that task will be spawned asynchronously, thus letting the main program continue its execution immediately. Otherwise, the original call is left untouched.

The task generation mechanism is essential for the asynchrony of the model: the main program can keep going while tasks are spawned and processed in the background by the runtime.

3.2.4.2 Data Access Surveillance

Sometimes, a piece of data that is created or updated by a task is later accessed from the main program, in any of the ways that have been described in Chapter 2, Sections 2.3.4 to 2.3.7. In such a situation, the main program cannot continue running until the right value for those data - the output of the task - is obtained.

In that regard, the Javassist tool permits to intercept and get information about several kinds of useful events, namely: method calls (to inspect their callee objects or parameters), field accesses (to check the associated object), object creation (to control streams created on files) and accesses to an array element (to check the array). This wide variety of supported pointcuts/events makes it possible to leave all the burden of watching data accesses to the runtime and, consequently, to allow the programmer to use her data in the main program in a normal way: it is the code inserted during the instrumentation phase that takes care of synchronisation.

From the moment a piece of data is first accessed by a task, it remains under surveillance of the runtime, which maintains a registry of task data; this is important not only to discover dependencies between tasks, as will be seen in Section 3.3, but also to control data accesses from the main program. Indeed, the runtime must watch such accesses in order to guarantee the sequential memory consistency of the application. Therefore, when the main program reaches a point where some data previously created/modified by a task is accessed, the runtime detects such situation, blocks the thread running the main program and starts a synchronisation to fetch the data from the node where the task ran. Once the data are available, the access is performed and the main program resumes its execution.

3.3 Data Dependency Analysis

As a result of the instrumentation phase, the Java StarSs runtime produces a modified bytecode of the application that can determine when a selected method or service is invoked and, in such a situation, instruct the runtime to asynchronously create a task.

Consequently, as the main program runs, the runtime receives task creation requests. Each of these requests contains information that uniquely identifies and describes the task, most of it taken from the task selection interface and its annotations. Of special interest is the list of task parameters, which stores the value, type and direction of each parameter.

The information about parameter direction is of utmost importance for the runtime, because it is the basis for the data dependency analysis system. A

task is said to be dependent on another task if the former reads some data written by the latter. In this sense, for every new task, the runtime detects the data dependencies between that task and the previous ones, taking into account how all of them access their parameters. As new tasks come and their dependencies are discovered, the runtime dynamically builds a *task dependency graph*, whose nodes are the tasks and whose arrows symbolise the dependencies. Such graph represents the workflow of the application and imposes what can be run concurrently and what cannot.

Java StarSs features a complete dependency analysis mechanism, comprising all the data types that can be used in a Java program and that may be subject to dependencies, namely objects, arrays and files. Primitive types are excluded from this analysis because their direction is always IN, as discussed in Section 2.3.6 of Chapter 2, and consequently they cannot incur in dependencies. In order to know if two tasks access the same object or array, the memory references of these objects/arrays are compared, whereas in the case of files the absolute paths to the files are checked. It is worth pointing out that callee objects and return values, even if they are not strictly parameters, are also considered in the dependency study.

Figure 3.3 illustrates the dependency analysis technique of the runtime with an example. In Figure 3.3(a), there is the code of an application that generates random numbers and cumulatively adds them. Line 1 creates a Sum object - initialised to zero - that will store the sum, and line 2 sets a name for a file rdFile where random numbers will be written in. From line 3 to 6, a loop with two iterations calculates the sum of two random numbers. First, in line 4, genRandom generates a random number and writes it in rdFile; second, in line 5, the add method adds the number in rdFile to the value held by object sum.

Both genRandom and add are selected as a task in Figure 3.3(b). The file parameter of genRandom has OUT direction (it is truncated and overwritten with a new number at each iteration). add receives an input parameter of type file; moreover, it is an instance method of class Sum, and so it is always invoked on a callee object belonging to that class; such callee, as explained in Section 2.3.4 of Chapter 2, is assumed to be accessed in INOUT mode by default.

Finally, Figure 3.3(c) shows the task dependency graph built by the runtime as a result of running the code in Figure 3.3(a), taking into account the interface in Figure 3.3(b). A total of four tasks are created, two per iteration of the loop. The task numbers correspond to the order in which the tasks are generated. The continuous arrows represent the real dependencies, i.e. those that cannot be avoided, which appear when some data is first written by a task and later read by another task. In the application, such situation happens in two cases: first, when add reads the file written by genRandom in the same iteration; second, when add reads the accumulated value of sum, updated by another add in the previous iteration. Note how these dependencies are found automatically and on-the-fly as the application executes and the tasks are asynchronously spawned.

Section 3.4 will discuss another functionality of the runtime, data renaming, which is strongly related to the dependency analysis. Such functionality allows to prevent 'false dependencies', as it will be explained next.

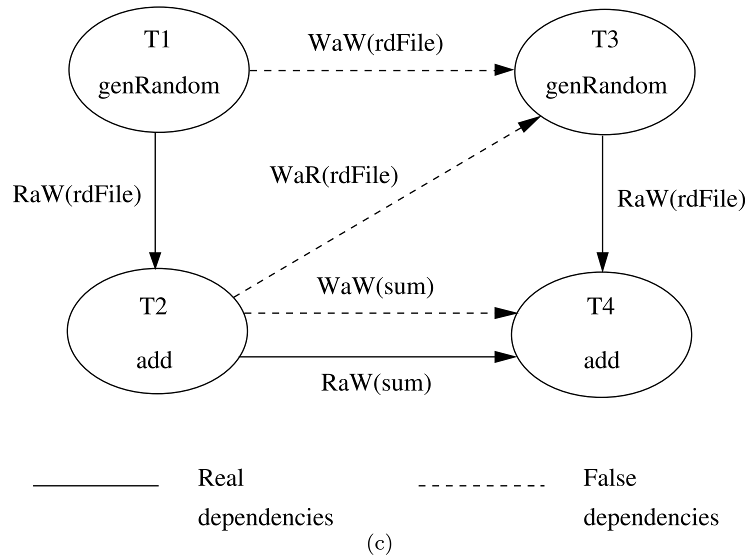
```
1 Sum sum = new Sum(0);
2 String rdFile = "random.txt";
3 for (int i = 0; i < 2; i++) {
4     genRandom(rdFile);    // rdFile ← random()
5     sum.add(rdFile);      // sum ← sum + val(rdFile)
6 }
```

(a)

```
public interface SumItf {
    @Method(declaringClass = "example.Sum")
    void genRandom(
        @Parameter(direction = OUT, type = FILE)
        String fileName
    );

    @Method(declaringClass = "example.Sum")
    void add(
        @Parameter(direction = IN, type = FILE)
        String fileName
    );
}
```

(b)



(c)

Figure 3.3: Main program of the Sum application (a), its corresponding task selection interface (b) and the graph generated when running it (c). At every iteration, the genRandom task method generates a random number and writes it in file rdFile; after that, method add (also a task) adds that number to a sum stored in the sum object. When executing the application, the runtime detects different kinds of dependencies, some of which can be avoided by means of a data renaming technique (WaW, WaR), whereas some cannot (RaW).

3.4 Data Renaming

Two subsequent accesses to the same data can lead to different kinds of dependencies [123]. When the first access is for writing and the second one is for reading (Read-After-Write or RaW) the dependency cannot be prevented. On the contrary, when dealing with Write-after-Write (WaW) and Write-after-Read (WaR) combinations, the dependencies disappear if the data are renamed.

In that regard, the Java StarSs runtime implements a renaming/versioning system for task data: when it processes a task that writes a parameter, the runtime generates a renaming - a new name - for that parameter. The main objective of the renaming technique is to avoid false dependencies between tasks, so that the task dependency graph exhibits more parallelism.

A renaming can be seen as a particular version of a given piece of data, and therefore it is identified by a pair <*data id, version id*>. In the runtime, new data versions are registered as writer tasks are generated; also, if the main program updates some data previously accessed by a task, a renaming is created as well. Hence, a registry is maintained to keep track of all the data accessed by tasks and the main program, as well as the versions they read and/or write. At every moment, the runtime knows which versions of which data each task needs/produces; furthermore, if the main program accesses some data, the runtime can identify the right version to obtain in order to guarantee the sequential memory consistency of the application.

Nevertheless, the renaming mechanism does not only involve creating a new name, it also implies allocating new memory/disk space for the renamed data. This way, two tasks that work with different versions of the same data can run simultaneously, even in the same node. For instance, in Figure 3.3(c), T2 and T3 can execute in parallel, since there are no real dependencies between them. T2 reads the first version of rdFile, produced by T1; on the other hand, T3 writes rdFile, generating a new version and causing a WaR dependency with T2. The fact that the new version of rdFile is stored in a new (renamed) file makes it possible that T2 and T3 execute concurrently, each working with a different renaming of rdFile. The explanation for the resolution of the WaW dependencies in Figure 3.3(c) is analogous.

As the execution progresses, a renaming might become obsolete. This happens when all the reader tasks for that concrete renaming have already completed. The runtime is able to identify such a situation and to instruct the node that stores the renaming to either delete it or reuse its space for a new version, in order to decrease the memory footprint. In the example of Figure 3.3(c), when T2 ends, the first version of rdFile becomes obsolete and can be removed because no other task will read it.

3.5 Data Layout and Transfer

As introduced in Chapter 2, Section 2.3, the creation of data in a Java StarSs application can happen in two scopes:

- In the main program: the data are allocated in the resource running the main program, either in memory (for objects, arrays and primitive types) or in a disk accessible from that resource (for files).

- Inside a task: tasks can also produce new data and make them available to possible consumers - other tasks or the main program - through their output parameters, e.g. a return value of type object/array/primitive or an output file parameter. In the case of method tasks, those data initially reside in the resource where the task ran, also in memory or disk. On the contrary, the data produced by a service task invocation is returned to the master runtime, which acts as the client of the service, and therefore those data are allocated in the main resource.

Data-allocating method tasks bring two advantages that are crucial for scalability. First, they permit to work around the restriction that all the application data must fit in one node (the one that executes the main program): the overall memory/disk space becomes the addition of every node's memory/disk. Second, the fact that data is initially distributed reduces the startup time of the application, since it prevents the main node from being the source of all data and turning into a communication bottleneck.

No matter their initial location, data will flow between nodes during the application execution depending on task scheduling, which will be described in Section 3.6. The runtime keeps track of all the locations where each data version is at every moment. This way, once the destination resource for a method task has been decided, the runtime knows where to find the input data of the task and can perform the necessary data transfers to that resource; concerning service tasks, the inputs are transferred to the main resource so that they can be embedded in the invocation message (more information in Section 3.7). Similarly, when the main program accesses some data that is not locally available, the runtime responds by transferring the right data version for the access to complete. In order to learn about the technologies used for transferring data, please refer to the infrastructure chapters (4, 5 and 6).

Hence, for the sake of simplicity of the programming model, the user is unaware of the exact location of her data and where they are transferred to as the application progresses. Also importantly, the user does not control the data layout of the application, e.g. by specifying the partitioning and distribution of an array among a set of resources when creating it. Nevertheless, the model does provide a basic mechanism to uniformly distribute data across worker resources: by marking data-allocation tasks as initialisation tasks, as was presented in Section 2.2.1 of Chapter 2 and will be further discussed next in Section 3.6.

3.6 Task Scheduling

A task that depends on other tasks remains in the task dependency graph until all its predecessors have completed and, as a consequence, its dependencies are solved. When this happens, the task is ready to be scheduled on a resource.

As a general rule, the runtime is provided with a list of worker resources, each of them having a number of assigned slots; these slots correspond to the maximum number of simultaneous tasks that the resource can execute. A computational resource usually has as many slots as cores, while in the case of a service resource that number is related to the server capacity. An example of the configuration files that describe the resources to be used by the runtime can be found in Appendix B.

If the runtime is able to find an available resource - with free slots - that can run a dependency-free task, such task is scheduled on that resource. Otherwise, the task is added to a queue of pending tasks waiting for a resource to be freed.

Java StarSs implements several scheduling policies to map a task to a given resource. These policies depend on the task type and they are discussed in the next two subsections.

3.6.1 Method Tasks

Method tasks are scheduled on computational resources, either physical or virtual. For such kind of resources, the runtime has information about their capabilities (e.g. memory, disk, architecture, etc.). As explained in Chapter 2, Section 2.2.1, the user can add an annotation to a method task in the task selection interface to define the constraints of that task. If such annotation is present for a given task, the runtime first filters the available resources depending on their capabilities, keeping only those that fulfill the task constraints; otherwise, the list of resources is not filtered.

Once the runtime has found the list of suitable resources for a task, it can apply two different scheduling algorithms, described next.

3.6.1.1 Algorithms

*** Locality Aware - Default**

As seen in Section 3.5, the runtime manages the flow of data between resources and maintains a registry of where those data reside. Such registry is a key structure for the default scheduling algorithm, which is locality-aware: the runtime tries to exploit data locality when selecting the worker resource that will execute the task.

More precisely, when deciding where to run a given task, the runtime checks its input/in-out parameters and where they can be found; the worker with a higher number of these parameters in its local memory/disk will be chosen. Similarly, when a task ends and a resource slot gets free, the pending tasks will be examined to find the one with the highest score (number of parameters already on that resource).

*** Round Robin - Initialisation Tasks**

Section 2.2.1 of Chapter 2 showed how the user can mark a task as an 'initialisation task'. These kind of tasks are treated differently in terms of scheduling: they are assigned to resources in a round-robin fashion.

Typically, initialisation tasks enclose some data allocation and initialisation instructions. This way, they can be utilised to uniformly allocate data across a set of resources, where those data can remain for later use.

Figure 3.4 illustrates the use of initialisation tasks and the corresponding scheduling decisions made by the runtime. In Figure 3.4(a), A is declared as a 4D matrix divided in $N \times N$ blocks, each block being a 2D matrix of doubles. The block creation is done inside the method createBlock, which allocates a block of $M \times M$ doubles and initialises every double with a given value VAL. createBlock is chosen as an initialisation task in Figure 3.4(b). When running the piece of code in Figure 3.4(a), assuming there are three available resources with four slots each and N=6, the allocation of the blocks among the resources resulting from the task scheduling would be the one in Figure 3.4(c).

3.6.1.2 Pre-scheduling

The Java StarSs runtime can work in pre-scheduling mode, where tasks are pre-assigned to resources with no free slots at that moment, so that the transfers for those tasks are triggered beforehand; later, when the processor gets free, the next task can be submitted immediately, without having to wait for any transfer.

This technique aims to overlap computation and communication as much as possible, as well as to distribute the load of the master runtime all along the execution, preventing 'hot spots' when many transfers have to be performed.

3.6.2 Service Tasks

Service tasks map to operations that execute on service providers. A certain service can be offered by more than one provider, that is, several instances of the same service can be deployed at different locations. The way the runtime schedules tasks on service instances depends on the kind of service invocation (see Chapter 2, Section 2.3.2).

On the one hand, for a stateless service invocation, the runtime simply picks one of the free service instances, i.e. with at least one slot available. Thus, when receiving a bunch of tasks linked to the same service, the load is balanced among the instances of that service.

On the other hand, stateful service tasks are necessarily tied to a particular instance of the service, because they modify the internal state of that instance. As a consequence, all the tasks resulting from stateful invocations of the same service are scheduled on the same service instance. In the task dependency graph, all those tasks are arranged in a chain of dependent tasks to ensure mutual exclusion when updating the state of the instance. Therefore, none of the tasks is scheduled until the previous one in the chain has finished.

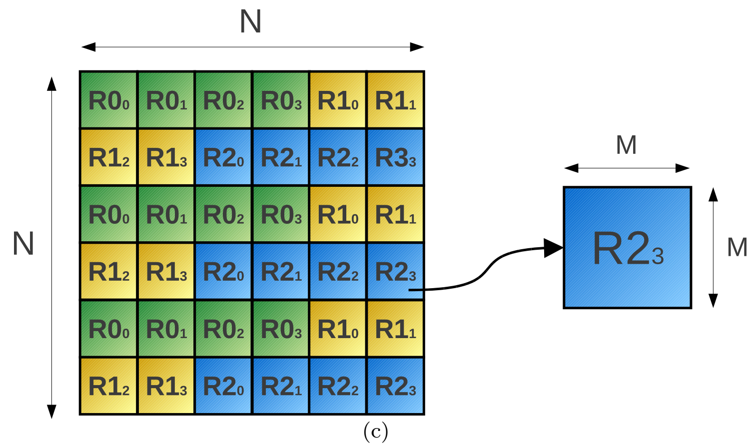
```
double[][][][] A = new double[N][N][][];

for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
        A[i][j] = createBlock(M, VAL);
```

(a)

```
public interface AppItf {
    @Method(declaringClass = "example.App", isInit = true)
    double[][] createBlock(
        int blockSize,
        double initValue
    );
}
```

(b)



(c)

Figure 3.4: In the code snippet in (a), A is a matrix divided in $N \times N$ blocks. The createBlock method allocates a single block of size $M \times M$ doubles and initialises all its elements with a given constant VAL. createBlock is selected as a task in the interface in (b) and is also marked as an initialisation task (isInit = true field in the @Method annotation); note that the parameters of createBlock do not need the @Parameter annotation, since their type is primitive and, consequently, their direction is IN. Finally, the scheduling of the createBlock initialisation tasks leads to the allocation of blocks among resources shown in (c), assuming 3 resources, 4 slots per resource and N=6.

3.7 Task Submission, Execution and Monitoring

Once a task is scheduled on a certain resource and the necessary transfers of its input data have been performed, the task is ready to be sent for execution.

In the case of a method task, the master runtime asynchronously submits the task to the destination resource chosen in the scheduling step, where the input data is already available; moreover, it registers for notifications coming from the worker resource to inform about the completion of the task. In the destination resource, the pre-deployed worker part of the runtime (see Figure 3.1) will be in charge of the task execution. Depending on the infrastructure, that worker runtime is transient or persistent: while in the former case a new JVM process is started every time a task request arrives, in the latter a process remains in the resource all along the application lifetime. The motivation and implications associated to each type of worker runtime will be further explored in Chapters 4, 5 and 6.

Concerning service tasks, the input data is in the main node, where the master runtime acts as the client and performs a synchronous invocation of the selected service instance. When the requested operation ends on the server, the runtime gets a response with the return value of the operation, if any.

Furthermore, for any kind of task, the master runtime implements a fault-tolerance mechanism: if there is an error in a task submission, the runtime tries a second time with the same resource; if the submission fails again, the task is rescheduled on another resource.

When a task completes normally, the runtime removes it from the task dependency graph, possibly resulting in newly dependency-free tasks that are ready to be scheduled.

Information about the underlying technologies used for task submission and monitoring can be found in Chapters 4, 5 and 6.

3.8 Summary

This chapter has provided an overview of the basic functionalities of the Java StarSs runtime (Figure 3.5), so that the user can get an idea of the whole execution process of the application: how the main program is instrumented to allow the creation of asynchronous tasks and to watch the data accesses; how the created tasks can incur in data dependencies, and how the runtime detects them and builds a task dependency graph; how the application data is renamed to prevent some types of dependencies and how the runtime keeps track of all the data versions and their locations; how dependency-free tasks are scheduled on a set of available resources, possibly considering data locality or task constraints; how the runtime transfers the input data of a task prior to its execution; how tasks are submitted to their destination resources and their completion is monitored.

All these functionalities are relevant because they free the user from managing data and computations in the context of concurrency and distribution. Of

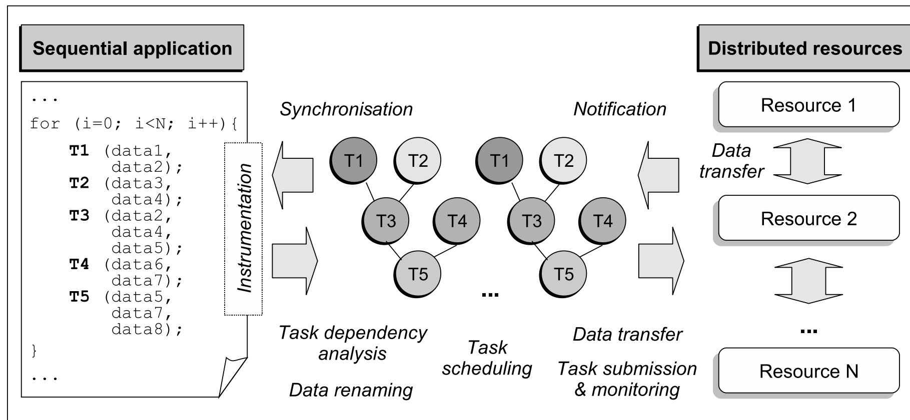


Figure 3.5: Overview of the basic features of the Java StarSs runtime.

special importance, though, are the instrumentation and dependency detection features, because they are crucial for enabling the programming model presented in Chapter 2. The fully-sequential programming would not be possible if the application were not automatically instrumented, and both task asynchrony and data access from the main program inevitably require a dependency analysis mechanism.

In this sense, the Java StarSs runtime integrates a programming model based on asynchronous computations with an exhaustive data dependency detection and synchronisation system, which comprises all the data types that can be used in a Java application, namely objects, arrays, primitive types and files. Furthermore, the runtime is able to handle two kinds of task, regular methods and service operations, and make possible the data exchange between them.

The runtime features described here are common to all the infrastructures contemplated in this dissertation. The next three chapters will go into detail about how these features were designed and implemented in each infrastructure, considering the singularities of the Grid, Cluster and Cloud scenarios while abstracting the programming model from them. Those chapters will also evaluate both the programmability offered by the model and the results of execution tests in every scenario.

Chapter 4

Grid

This chapter starts a trilogy of chapters that overview three kinds of parallel and distributed infrastructures, corresponding to the different scenarios where the programming model has been applied. Amongst all the model's features presented in Chapter 2, each infrastructure chapter will highlight, demonstrate and evaluate those features that are most relevant in each scenario.

In particular, the current chapter will begin this three-stop journey focusing on *the Grid*. The content organisation will follow the same pattern in every infrastructure chapter: first, an introduction to the context and to some basic concepts; second, an explanation of the runtime design decisions motivated by the scenario; third, a description of the technologies that influenced the runtime implementation for that infrastructure; fourth, a programmability evaluation of the programming model, comparing it to another approach for the same scenario; fifth, the results of the experiments carried out in the infrastructure; finally, a related work section and a concluding summary.

4.1 Context

4.1.1 The Grid

The term *'Grid'* was coined by Ian Foster and Carl Kesselman back in the late 1990s [128] and it designs a set of loosely-coupled and *heterogeneous resources*, owned by multiple parties, which are usually scattered over a *wide geographic area* across multiple administrative domains and which share their computing power and data storage capacity.

The origins of Grid computing lie in the growing need of certain scientific applications for computing and storage capabilities. Grids can combine the resources of many computers to create a vast computing resource, which can be used to accomplish large and complex tasks.

Although the Grid was initially envisioned as a global network of computers joined together, reality has brought instead hundreds of grids around the world, each one built to help one or more specific groups of users.

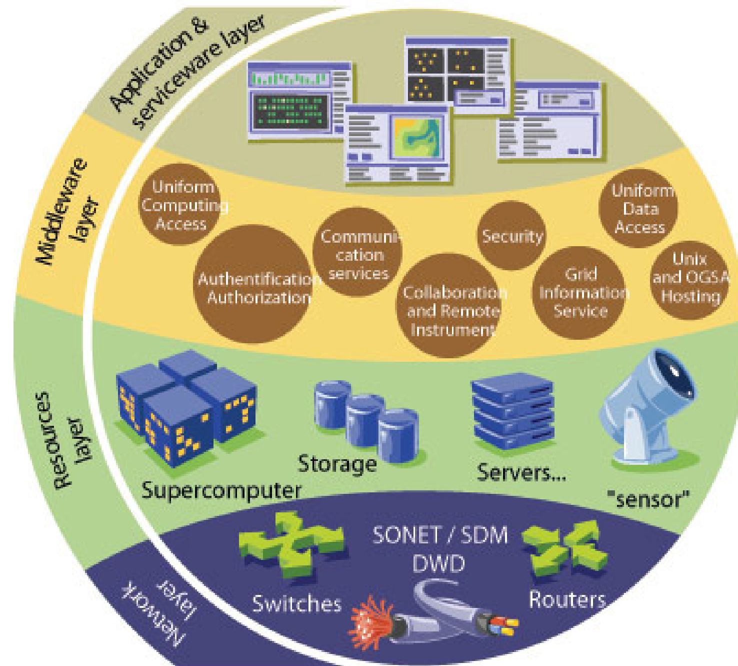


Figure 4.1: Grid architecture layers. Courtesy of the GridCafé website [23].

4.1.1.1 Architecture

The architecture of a grid is generally divided in four layers, shown in Figure 4.1. While the two uppermost ones focus on the user, the two lowermost ones are centred on the hardware.

- *Network*: it ensures the connectivity of all the resources of a grid. Some grids are built on top of dedicated high-performance networks, such as the intra-European GÉANT network [62], but grid nodes can also be interconnected by slow Wide Area Networks (WAN) or the Internet.

- *Resources*: this layer contains computers, storage systems, sensors and instruments like telescopes that are connected to the network.

- *Middleware*: the middleware layer brings together the elements located in lower layers (servers, storage, network) and enables them to participate in a unified Grid environment. Furthermore, it provides the applications on the top layer with access to Grid resources, by means of services covering

the submission of jobs (computations), data management, security and information systems. Just like Grid resources, Grid middleware is heterogeneous too: each Grid infrastructure is managed by some given middleware with its own set of tools, for instance the Globus Toolkit [117], gLite [132] or UNICORE [163].

- *Applications*: applications in science, engineering, business, finance, etc. fall into this layer. Moreover, it also includes portals and development toolkits to support the applications, as well as software that monitors resource utilisation by Grid users.

4.1.1.2 Virtual Organisations

Grid computing is about people sharing their resources to achieve a common goal. This leads to the concept of a *Virtual Organisation* (VO): a group of people in different organisations seeking to cooperate and share resources across their organisational boundaries [118].

Therefore, the users of a grid are grouped into VOs. In order to achieve their mutual objective, people within a VO agree to share their expertise and resources, thus contributing to the creation of a grid. This grid can give VO members direct access to each other's computers, programs, files, data, instruments and networks.

Nevertheless, such resource sharing must be controlled. Members of a VO are subject to a set of resource-usage rules and conditions, which establish the resources to which they have access and in what amount.

4.1.1.3 Secure Access

A subset of the Grid middleware is responsible for guaranteeing the secure access of users to resources. One of the aspects of Grid security is authentication of both users and resources.

The Grid utilises asymmetric cryptography [122] for authentication. Any user willing to access the resources of a grid needs to be in possession of a key pair: a public key or *certificate*, which is made public, and a private key, which is kept secret. The user obtains a valid key pair from a Certification Authority (CA), an entity that issues digital certificates and that is trusted by that grid; the CA signs the key pair to confirm the identity of the subject - the user - in the certificate. A CA-signed key pair is also known as the *credentials* of a user.

Once a user has got a certificate, she can use it to authenticate herself when requesting access to Grid resources. Many grids use, in addition, a system of proxy certificates or *proxies* for a user to delegate her rights to Grid services, which may have to contact other services in behalf of the user. A proxy is created by the user from her own credentials and is only valid for a limited period of time, in order to minimise potential damage should the proxy be compromised.

Proxies can contain extensions carrying additional information about users, such as their affiliations with VOs. This way, users are only authorised to access Grid resources in a manner determined by their VO membership.

4.1.1.4 Data Management

Data management is one of the key features of a grid, where potentially large amounts of data are distributed amongst remote sites, possibly all over the world. The primary unit for Grid data management is the *file* [19]: big data is stored in files that may be replicated across different sites for reliability and faster access. To ensure consistency between replicas, typically Grid files are read-only, i.e. not modified after creation. Moreover, users do not need to know where a file is located, as they can refer to a set of replicas of a file with a logical name.

Storage devices - e.g. disks, tapes - are connected to other resources - computers, instruments - by means of the network layer. On the other hand, the middleware layer provides a set of tools to manage the content of those storage devices, allowing to:

- Copy files between data stores residing at distributed sites. The tool of choice here is GridFTP [77], a high-performance secure data-transfer protocol implemented by the Globus Toolkit.
- Uniquely identify Grid files, mapping logical file names to physical file locations. This is achieved with a Replica Catalogue for naming and locating replicas.
- Combine file transfer with file cataloguing and offer it as an atomic transaction to the user (Replica Management Service).

Usually, the input files required by a Grid job are staged-in (transferred) from persistent storage to some temporary path in the node where the job is scheduled. Similarly, when the job ends, the generated output files are staged-out from that node to some long-term storage element. Jobs are normally *coarse-grained*, lasting from several minutes to hours, in order to compensate the overhead and latencies of a Grid environment (middleware processing, movement of data to distant resources, queue time).

4.1.2 e-Science Applications

As introduced earlier in this chapter, behind the inception of Grid computing there is the growing need of some scientific applications for massive computation and storage capabilities. This computationally intensive science working on immense data sets was named *e-Science*, and nowadays it is perhaps the most important field on which Grid technologies are applied.

Scientists from around the globe can use the Grid to tackle bigger problems, to enable projects that would be impossible otherwise. Besides, e-Science is about global collaboration in key areas of science: scientists can share data, data storage space, computing power, expertise and results in an unprecedented scale.

The following points are examples of scientific areas that have evolved into e-Science [23], [19], [71]:

- *Particle physics*: the Large Hadron Collider (LHC) is the world's largest particle accelerator; it was built by the European Organisation for Nuclear Research (CERN) to test the predictions of different theories of particle physics and high-energy physics. Annually, the LHC generates 15 Petabytes of data, whose processing requires huge computational and storage resources, as well as the associated human effort for operation and support. To help with that, the Worldwide LHC Computing Grid (WLCG) was created: a grid integrated by thousands of computers and storage systems in hundreds of data centres worldwide.

- *Bio-informatics*: over the last years the size of biological sequence databases has grown exponentially, now containing millions of genes and proteins that are freely available to researchers over the Internet. In order to efficiently process and analyse these biological data, many tools have been developed; for example, programs that locate a gene within a sequence, predict a protein's structure or function and cluster protein sequences into families of related sequences. Grids provide the infrastructure to perform such kind of analysis in a reasonable time scale.

- *Earth sciences*: earth science applications and, more precisely, weather and climate modeling, are currently among the most computationally-demanding programs. Furthermore, a number of tasks such as ensemble prediction, sensitivity analysis, etc. consist of running the same application many times with slight variations in the configuration parameters (which is known as parameter-sweeping algorithms), thus requiring an appropriate production infrastructure like a grid.

This chapter will show examples of how the Java StarSs programming model and execution runtime can help developing e-Science applications and running them over large-scale heterogeneous grids.

4.1.3 Grid APIs: Standardisation Efforts

One of the reasons why the romantic idea of a worldwide Grid - seen as a single, interconnected and interoperating computer farm - has not become real is the considerable lack of widely-adopted Grid standards. Many smaller grids exist instead, each customised to meet the specific needs of a user group, each using a certain set of technologies.

This issue also affects the development of Grid applications. A programmer willing to access Grid services using Application Programming Interfaces (APIs) faces several problems: first, due to middleware heterogeneity, applications implemented with a given API are not portable to other Grid sites managed by different middleware; second, Grid APIs tend to be too low-level and verbose for programmers who are domain experts rather than computer scientists, which hampers their adoption; third, in some cases the APIs change too frequently for applications to follow.

In the Grid world, the Open Grid Forum (OGF) [42] is the largest community pursuing the adoption of Grid standards. The OGF provides an opportunity for volunteers to contribute to the development of new standards. In this sense, there is an OGF working group named SAGA [120] (Simple API for Grid Applications), which targets the definition of a uniform API for Grid applications to access common Grid services, such as job management, file transfer and security. This API aims to be high-level, abstracting the programmer from the middleware underneath. The design of SAGA was very much influenced by previous work on the Grid Application Toolkit [78], a technology that is used by the Java StarSs runtime and that will be further explained in Section 4.3.2.

4.1.4 Component-Based Grid Software

Along with the efforts to simplify and standardise APIs for Grid services, a complementary initiative appeared with the purpose of facilitating the development of Grid software, ranging from applications to middleware: *Component-Based Software Engineering* (CBSE). Although the idea of using components in software is not new [139], it regained interest in the late 1990's - early 2000's [96] [131] due to, primarily, the growing complexity of software systems and the inability of the extant programming models to face that issue properly.

CBSE changes the way of developing software systems, composing rather than programming them. A component is a unit of independent deployment and composition that states, by means of interfaces, the services that uses and provides. By creating software from off-the-shelf components, CBSE promotes *reusability*: one can benefit from the functionalities of already existing components, thus reducing development time.

The manner in which components are composed and interact is specified by a component model. As a part of the CoreGRID Network of Excellence [61], a working group was created to define a component model particularly intended for the Grid: the *Grid Component Model* (GCM) [74]. The main objective of GCM is to hide to the programmer the inherent complexity of a large-scale distributed environment like the Grid: heterogeneous hardware and operating systems, user authorisation and security, resource load and failure, etc.

The following points summarise the main properties of GCM:

- *Hierarchical organisation*: GCM components can be either primitive or programmed as compositions of other components (composites). Primitive components encapsulate basic functionalities of a software system, whereas composites group related functionalities.

- *Functional and controller interfaces*: used to access the functionalities implemented by the component and to dynamically reconfigure the component by modifying its behaviour, respectively.

- *Structured communications*: support for one-to-one communications between client/server interfaces, as well as collective interactions such as multicast (one-to-many) and gathercast (many-to-one).

- *Autonomic managers*: support for autonomic behaviour of components - self-configuration, self-optimisation - necessary in highly-dynamic, heterogeneous and networked architectures.

- *Deployment*: the application code is free of details about deployment of components; instead, they are specified in separate descriptors.

The principles of CBSE and GCM were put in practice when designing the Java StarSs runtime, as will be seen next in Section 4.2.

4.2 Runtime Design

Chapter 3 provided a general description of the basic functionalities of the Java StarSs runtime. The way they are implemented depends, though, on the specificities of each infrastructure.

Therefore, the design and implementation of the Grid flavour of the runtime were driven by the characteristics of Grid computing, as well as by the technologies and initiatives in that field. At the light of what has been presented in Section 4.1, the list of Grid characteristics that influenced the genesis of the Grid runtime includes:

1. Geographically-distributed and heterogeneous resources: the runtime must be able to exploit resources with diverse hardware and software characteristics, possibly scattered around the globe and belonging to different administrative domains.

2. Heterogeneous middleware: the runtime must be able to utilise various kinds of middleware to access a basic set of Grid services, namely job management, data transfer and security management.

3. Resources bound to failure: Grid applications normally run for long periods of time and use a big amount of non-necessarily reliable resources, which increases the probability of errors at some point (e.g. node or network failure, software errors). Consequently, there is a need for the runtime to implement fault-tolerance mechanisms for the application to continue running even in case of failure of some kind.

4. The primary unit for Grid data management is the file: the runtime must be able to deal with the files accessed from a Grid application. Mainly, this involves registering those files and eventually transferring them between Grid resources.

5. Slow networks: the fact that Grid resources can be separated by thousands of kilometres and interconnected by wide-area links increases network latency and makes data locality more important. Moreover, when the size of the application data is big, avoiding transfers becomes even more crucial, especially when lacking a high-bandwidth network underneath.

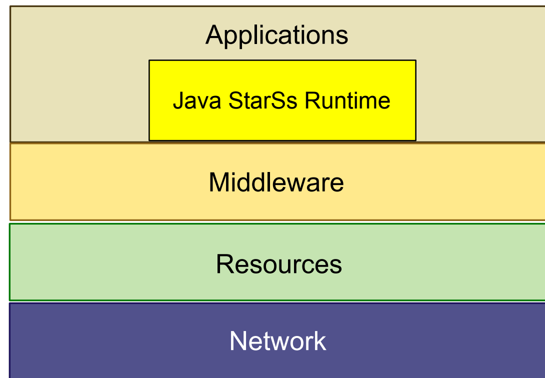


Figure 4.2: Location of the Java StarSs runtime in the Grid stack.

In the Grid stack, depicted in Figure 4.1, the Grid implementation of the runtime is located in the Applications layer, giving support to applications programmed in the Java StarSs model and interacting with the middleware in the lower layer, as shown in Figure 4.2. The next subsections will present and justify the design decisions for the runtime, taking into account the aforementioned Grid characteristics.

4.2.1 Componentisation

The Grid runtime was designed and implemented following the principles of CBSE and, in particular, GCM, introduced in Section 4.1.4 as a model to build componentised Grid software.

In a first phase of design, the runtime of GRID superscalar (GRIDSs) [85] was taken as a starting point. GRIDSs also offers a task-based dependency-aware programming model for Grid applications, and it can be considered the predecessor of Java StarSs (see Section 4.6 to learn more). The GRIDSs runtime was studied in order to identify its main functionalities, namely dependency analysis, scheduling and file and job management. Each of these functionalities was assigned to a separate component, thus resulting in a componentised runtime with a new set of interesting properties: reusability, ease of deployment in Grid contexts, flexibility and separation of concerns.

The components that form the Grid runtime are depicted in Figure 4.3. They inherit from GCM some of its properties, listed in Section 4.1.4: they are structured in a hierarchical way; they invoke each other through well-defined interfaces; they can have one-to-one or collective communications (e.g. a multicast initialisation call for all the components); the information to deploy them in a Grid infrastructure is contained in descriptor files.

The next subsections explain the functionalities that each of the runtime components enclose.

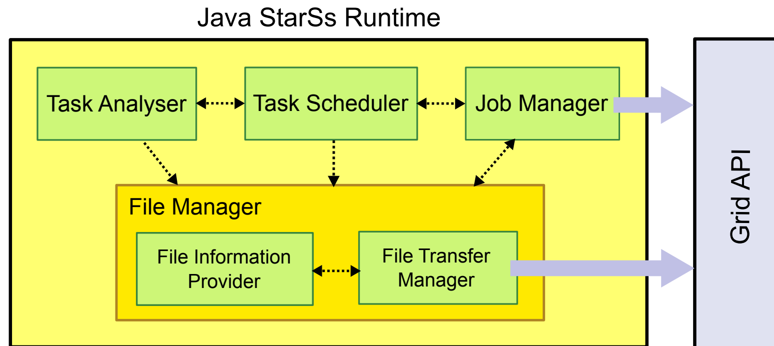


Figure 4.3: Component hierarchy and interactions in the Grid Java StarSs runtime, which sits on top of a uniform Grid API.

4.2.1.1 Task Analyser

It processes the tasks coming from the application and detects their dependencies, based on which files they access and how (read, write or both), building a task dependency graph. It interacts with the File Manager to register the file accesses of each task. Once a task is free of dependencies, the Task Analyser sends it to the Task Scheduler.

4.2.1.2 Task Scheduler

It decides where to execute the dependency-free tasks received from the Task Analyser. This decision is made by a scheduling algorithm with three information sources: first, the available Grid resources and their capabilities; second, the set of task constraints - if any - defined by the user in the task selection interface (see Chapter 2, Section 2.2.1); third, the location of the input data required by the task (information obtained from the File Manager) to exploit locality. This component addresses Grid characteristics #1 and #5.

4.2.1.3 Job Manager

It is in charge of job submission and monitoring. It receives the scheduled tasks from the Task Scheduler and delegates the file transfers required by each task to the File Manager. Upon completion of a task's transfers, the Job Manager submits the task to its target resource by creating a Grid job with a Grid API, performing the necessary user authentication; then, the component controls the proper completion of the job. The Task Scheduler and the Task Analyser are notified when a task ends, for them to update the list of available resources and the dependency graph, respectively. Finally, the Job Manager implements a fault-tolerance mechanism: failed jobs are retried first to the same resource and, if the error persists, the Task Scheduler is asked for a rescheduling of the corresponding task. This component addresses Grid characteristics #2 and #3.

4.2.1.4 File Manager

It takes care of all the operations where files are involved, playing the role of a replica management system. It is a composite component which encompasses the File Information Provider and the File Transfer Manager components. The former gathers all the information related with files: what kind of file accesses have been done, which versions of each file exist and where they are located, thus acting as a file catalogue; every time a task writes a file (out or in-out access), that new version is assigned to a new logical file and added to the catalogue. The latter is the component that actually replicates files from one resource to another, using a Grid API and also authenticating the user first; moreover, it informs the File Information Provider about new physical locations (replicas) of logical files. The File Manager applies fault-tolerance strategies too: each failed transfer is retried using all the available replicas for that file, and if the error repeats the Task Scheduler is asked for a rescheduling of the corresponding task. This component addresses Grid characteristics #2, #3 and #4.

4.2.2 Uniform Grid API

The Job Manager and File Transfer Manager components are the ones that communicate with Grid services, in particular for job creation, file transfer and user authentication. As discussed in Section 4.1.3, some efforts have been made to standardise and facilitate the invocation of Grid services by means of an API.

In this sense, the aforementioned components of the Java StarSs runtime interact with the Grid using a Grid API (Figure 4.3). Such API provides a set of methods to uniformly access various kinds of Grid middleware. This way, applications programmed with the Java StarSs model can run on grids managed by different middleware, in a transparent way to the user. Tasks generated by an application can be submitted to any of the grids available, and files can be transferred to/from any grid, or even between two grids. Furthermore, prior to any of those operations the user is automatically authenticated in that grid, provided that the user credentials are present in the host from where the Grid services are invoked (usually, where the application is launched).

4.2.3 Execution Model

Chapter 3, Section 3.1 explained how the Java StarSs runtime is divided in a master and a worker part. The component structure shown in Figure 4.3 corresponds to the master part, which encompasses the main functionalities of the runtime.

When the user starts the execution of the application, first of all the components of the master runtime are deployed and initialised. The way the components are deployed is specified in a deployment descriptor that, in short, maps the components to the physical resources where they will be created. Java StarSs provides a default deployment descriptor where all the runtime components are mapped to the user's host - where the application is launched - but it can be

modified to deploy any component in a remote host, thus benefiting from the distribution capabilities of GCM.

Once the runtime is up and running, its components begin to process the incoming application tasks as described in Section 4.2.1.

Although the application execution is mostly managed by the master runtime, there is still a need for a worker part that performs some task handling in the Grid execution resources. Prior to the execution of the application, that worker runtime must be installed in the Grid resources to be used, along with the bytecode, binaries, scripts, etc. that the application tasks may require.

In Grid environments, it is common to interact with a head node to which jobs are submitted; such node then delegates the scheduling of each job to a local scheduler in charge of managing a certain set of resources, all this being hidden from the user. In such a scenario, workers must be *transient*: no permanent process can be kept in a Grid resource because, on the one hand, the final execution node is unknown and, second, the master runtime cannot interact directly with that node. Instead, a new worker process is created for every job, which launches a JVM, parses the execution parameters received from the master, runs the task and then terminates.

4.2.4 Data Model

The Grid Java StarSs runtime focuses on files as the main unit of data. In the programming model, the user can write a Grid application that deals with files as described in Chapter 2, Section 2.3.7. These files can be either local or remote (e.g. located in some GridFTP server); in the last case, the user references them with a complete URI (Uniform Resource Identifier [66]), including the resource name and the absolute path of the file in the resource.

During the application execution, files are moved between Grid resources depending on the task needs and in a transparent way to the programmer. Typically, the steps that take place when a task has to be executed are:

1. After the master runtime has decided the target resource for the task, it transfers the task input files to a storage server or disk that is accessible from that target resource.

2. The job is submitted to the target resource and, as a result, the worker runtime ends up being launched in the final execution host, which is possibly behind a head (or front-end) node and selected by a local scheduler of the target resource.

3. Stage in: if the execution host is hidden to the master runtime, the task input data must be copied from a storage server to a temporary directory in the local disk of that host; this is done for faster access to the data. File stage-in is taken care either by the worker runtime or, in some cases, by the Grid middleware itself.

4. The task runs in the execution host, perhaps generating one or more output files. If the host is behind a front-end, these output files are normally placed in the same temporary directory as the input ones.

5. Stage out: once the task ends, its output files might need to be moved to a storage server, if they are in a hidden host's temporary folder (otherwise they would be lost). Again, this step can be the duty of the worker runtime or the middleware, and it is not necessary if the master runtime has direct access to the disk of the execution host.

The experiments in Section 4.5.1 will illustrate how data is managed in various grids controlled by different middleware.

As a final note, the fact the Grid focuses on files does not mean that the Java StarSs programmer cannot use objects or arrays in her application: the master and worker parts of the Grid Java StarSs runtime were extended to automatically serialise/deserialise objects and arrays to/from files, so that they can be sent to a Grid resource and passed to the tasks scheduled there.

4.3 Relevant Technologies

This section presents the Grid technologies that were used to implement the runtime design seen in Section 4.2.

4.3.1 ProActive

ProActive [99] is a Java Grid middleware library for parallel, distributed and multi-threaded computing. Among some other features that will be explored in other chapters of this thesis - more precisely, in Chapter 6 - ProActive provides the reference implementation of GCM.

Therefore, the components of the Grid runtime are built using ProActive, which offers an API to create, start, stop and destroy GCM components. The hierarchical component structure is defined in an ADL file (Architecture Description Language) [74], where components are described in terms of their interfaces, their bindings with other components and the Java classes that implement their functionalities (for primitive components). Individual communications as well as collective ones are supported for the component bindings.

As discussed earlier in Section 4.2.3, components can be mapped to resources by means of deployment descriptors, which are processed by ProActive when setting up a component structure. Section 4.5.2 will give an example of a custom remote deployment of the master runtime components.

4.3.2 The Grid Application Toolkit

JavaGAT is the Java version of the Grid Application Toolkit [78], which is a generic and flexible API for accessing Grid services from application codes, portals and data management systems. The calls to the GAT API are redirected

to specific adaptors which contact the Grid services, thus offering a uniform interface to numerous types of Grid middleware. Among the adaptors supported by JavaGAT there are widely-used middleware like the Globus Toolkit [117], gLite [132] or UNICORE [163], as well as an adaptor to establish SSH (Secure Shell) [63] connections. It is worth noting that, in order to invoke Grid services from a given machine with JavaGAT, it is not necessary to have all these middleware installed in that machine: the JavaGAT adaptors and the user credentials are enough.

The Java StarSs Grid runtime invokes the JavaGAT API to perform three basic operations:

- Authenticate a user in a grid: JavaGAT allows to define 'security contexts', which are containers for the credentials of a user in a given grid, e.g. a user certificate, a VO proxy or SSH keys.
- Submit and monitor Grid jobs: the master runtime encapsulates tasks into JavaGAT jobs when submitting them for execution. A JavaGAT job contains the following information: the program to be run (a script of the worker runtime), the parameters (related to the particular task) and the target resource. Moreover, the status of the job can be monitored, in order to control its proper completion.
- Transfer files: the master runtime uses JavaGAT to manage logical files, i.e. files that are identified by a logical name and that contain a set of replicas. When a task has a given logical file as input, it is copied to its destination from one of its replicas.

Section 4.5.1 will illustrate the use of the JavaGAT API in several real grids.

4.4 Programmability Evaluation

This section will evaluate the ease of programming of Java StarSs in Grid environments. For that purpose, a comparison with another approach in the same field will be carried out, more precisely with Taverna [142], a well-known graphical tool for designing and executing Grid workflows.

The comparison study will consist in implementing the same bio-informatics application with both Java StarSs and Taverna, and then highlighting the most relevant differences. The next subsections will present Taverna and the application and finally conclude with the comparison discussion.

4.4.1 Taverna

Taverna mainly differs from Java StarSs in the fact that applications are developed graphically rather than programmatically. Despite this fact, it has been chosen for this study because of its popularity, especially in the life-sciences area.

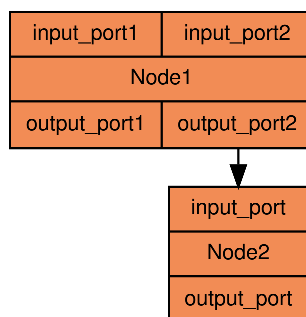


Figure 4.4: Simple workflow in Taverna. Node1 has two input ports and two output ports, while Node2 has only one of each kind. The link between the two nodes represents a data dependency.

A Taverna workflow is specified by a directed graph where nodes represent software components. A node consumes data that arrives on its input ports and produces data on its output ports. Each arrow in the graph connects a pair of ports, and denotes a data dependency from the output port of the source node to the input port of the destination node. Figure 4.4 shows a very basic workflow example.

The nodes of a Taverna workflow can be computations executed in the Grid, their ports being their input/output files or streams. Although the official Taverna distribution only includes an SSH adaptor to submit computations to Grid resources, some projects have developed plugins to make Taverna work on top of Globus-based middleware as well. Moreover, Taverna also supports external Web Services as workflow nodes.

Other features of Taverna include control flow structures, both implicit (like loops on output ports that are lists) and explicit (like condition checking encapsulated in a workflow node).

4.4.2 Hmmpfam Application

HMMER [24] is a bio-informatics suite that contains several tools for protein sequence analysis. It is based on profile hidden Markov models (profile HMMs) [108], which are statistical models of multiple sequence alignments. Each HMM represents a protein family, and captures position-specific information about how conserved each column of the alignment is and which residues are likely to occur.

One of the most important programs in the HMMER collection is *hmmpfam*. This tool reads a sequence file and compares each sequence in it, one at a time, against a database of HMMs, searching for significantly similar sequence matches with each model and producing a result file that summarises the best scores. The work performed by hmmpfam is computationally intensive and embarrassingly parallel, which makes it a good candidate to run on the Grid.

4.4.3 Comparison

4.4.3.1 Hmmpfam in Java StarSs

A common way of parallelising sequence-database search algorithms like hmmpfam is segmentation, which can be of two types:

- Query segmentation: it consists in splitting a set of query sequences so that each resource is responsible from a fraction of these query sequences. On the other hand, a copy of the database is replicated in every resource. As a result, several searches can be run in parallel.
- Database segmentation: in this case, independent fragments of the database are searched on each resource. Splitting the databases is becoming more and more necessary due to their increasing size: if the database does not fit in memory, the overhead of disk I/O can significantly hinder performance.

In the Java StarSs implementation of hmmpfam, both segmentation strategies are supported. The application is a completely sequential Java code that receives these parameters: database file, sequences file, name of result file (score report), command line arguments for hmmpfam, number of database fragments and number of sequence fragments. The code is divided in three main phases:

- *Segmentation*: the database file and the query sequences file are split, depending on the number of database fragments ($N \geq 1$) and the number of sequence fragments ($M \geq 1$), respectively, both received as parameter.
- *Execution*: the hmmpfam binary is wrapped in a Java method and invoked for each pair of sequence-database fragments ($N \times M$).
- *Reduction*: the partial outputs for each pair of sequence-database fragments are merged into a final result file. This is done by invoking a merge method that combines two partial results at a time, resulting in $(N \times M) - 1$ calls to that method.

Following the steps detailed in Chapter 2, Section 2.1.1, the potential application tasks were identified. Concerning the segmentation phase, it was not worth to select any method as a task because splitting the input files can be done fast enough locally. Regarding the execution phase, there was a clear candidate: the method containing the call to hmmpfam, which is the computationally intensive part of the application. Finally, the reduction phase was intended to be performed distributedly as a set of tasks - merge tasks - instead of locally and sequentially; if the merge phase were done entirely in the master node, the master would have to wait for the generation of the partial results, get them all and process them, which can cause a bottleneck when $N \times M$ is high.

The task selection interface resulting from identifying the application tasks is depicted in Figure 4.5. Two tasks are selected: first, `hmmpfam`, which calls the hmmpfam tool with some command line arguments for a given pair of input database-sequence fragments, and outputs a report file; second, `merge`, which

```
public interface HMMPfamItf {

    @Method(declaringClass = "worker.hmmer.HMMPfam")
    void hmmpfam(
        String clineArgs,
        @Parameter(type = FILE)
        String dbFragFile,
        @Parameter(type = FILE)
        String seqFragFile,
        @Parameter(type = FILE, direction = OUT)
        String reportFile
    );

    @Method(declaringClass = "worker.hmmer.HMMPfam")
    void merge(
        @Parameter(type = FILE, direction = INOUT)
        String report1,
        @Parameter(type = FILE)
        String report2
    );

}
```

Figure 4.5: Task selection interface corresponding to the Hmmpfam application in Java StarSs.

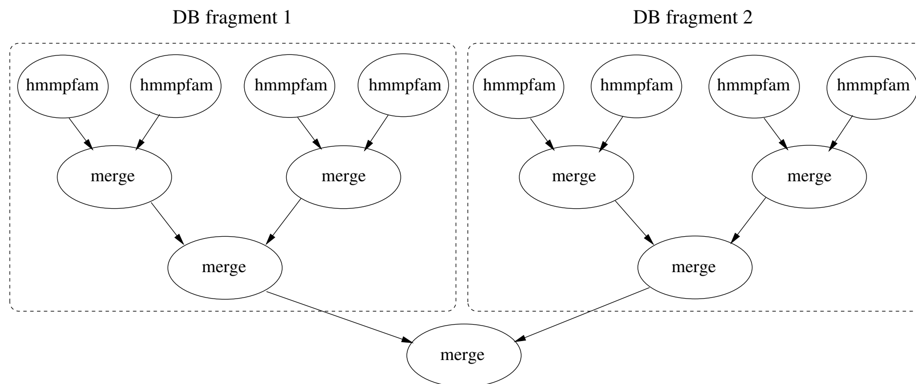


Figure 4.6: Example of a task dependency graph generated by Hmmpfam when running it with Java StarSs. In this case, the database is split in two fragments and the query sequences file in four parts. This creates eight independent tasks that run hmmpfam on a pair of database-sequence fragments. After that, there are three levels of reduction tasks, the last one merging the results from the two different database fragments.

receives two partial hmmpfam results and combines them into the first result file. As explained in Chapter 2, Section 2.2.2, if the direction is not specified it is assumed to be IN; on the other hand, the clineArgs parameter does not need to be annotated, since its type (object) and direction (input) are automatically inferred. For space reasons, the main program of Hmmpfam is provided separately in Appendix A.1.

At execution time, Hmmpfam generates a graph like the one in Figure 4.6. Note how N and M, which in this case are 2 and 4 respectively, determine the number of tasks and consequently the amount of parallelism exhibited by the graph.

In conclusion, the Hmmpfam application was designed with parallelism in mind - the problem is first split in independent computations and then the partial results are merged - but, most importantly, the programming is sequential, i.e. the user does not have to deal with the burden of parallelisation (spawning of asynchronous tasks, synchronisation, data dependencies).

4.4.3.2 Hmmpfam in Taverna

The structure of the Taverna implementation of Hmmpfam is quite the same as for Java StarSs. In this case, though, the application is composed graphically, the tasks being nodes of a Taverna workflow.

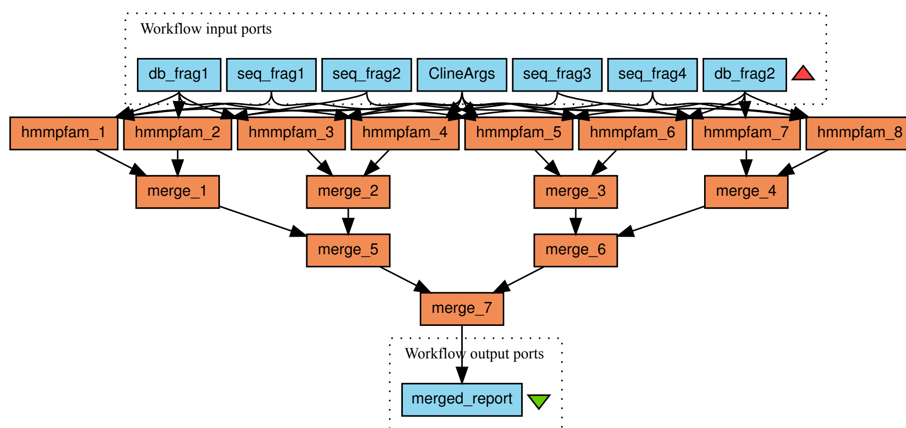


Figure 4.7: First version of Hmmpfam in Taverna.

Two distinct implementations of Hmmpfam in Taverna have been considered. The first one (Figure 4.7, shown with no details of the node ports for simplicity) is a workflow that receives as input the command line arguments and the fragments of database and sequence, previously generated; this appears in Figure 4.7 as 'Workflow input ports'. These inputs are passed to a first row of hmmpfam nodes, where one node was drawn for each pair database-sequence.

The output of the hmmpfams is connected to a set of merge nodes, which finally converge in a final report in 'Workflow output ports'. Therefore, this option requires to create statically a number of fragments and link them with an equally static number of hmmpfam nodes.

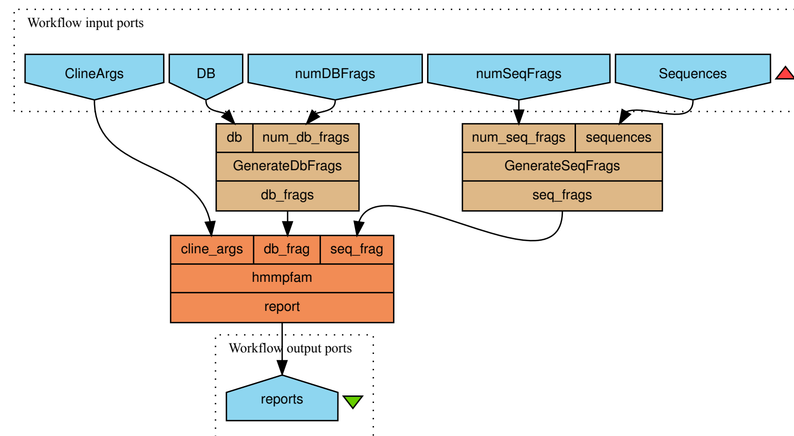


Figure 4.8: Second version of Hmmpfam in Taverna.

The second version (Figure 4.8) tries to bring some dynamism to the application. In this case, the workflow input ports contain the database file, the sequences file and the number of fragments for both. These data are passed to a couple of nodes, GenerateDbFrags and GenerateSeqFrags, which correspond to local processes; those nodes will produce each a list of database and sequence fragments, respectively. This is where Taverna's implicit iterators enter the picture: the two lists are connected to a third node, hmmpfam, which actually will be transformed into several nodes at execution time; for every two elements of the lists, a hmmpfam will be created, thus producing a list of output reports. Nevertheless, the downside is that the list of output files cannot be merged into a single file, Taverna does not provide any graphical mechanism to do that. Consequently, this option is simpler but it requires a post-processing of the list of partial reports, which is directly included in the workflow output ports.

4.4.3.3 Discussion

In light of the Hmmpfam implementations just described (Java StarSs and Taverna), this section highlights the main differences between these two approaches:

- *Graphical versus code programming*: Taverna users build the application workflow as a graphical composition of nodes. Contrarily, Java StarSs generates the task graph (analogous to the Taverna workflow) at execution time, as invocations to the selected tasks are found. Taverna also requires

some code programming, but only to fill the content of the nodes: the workflow drawn by the user constitutes the structure of the application.

- *Data dependencies*: Taverna users must know both how a node accesses its data (defined by input or output ports) and which are the data dependencies between nodes (defined by the lines that link nodes). In contrast, Java StarSs only requires the user to know how a task accesses its data, which is specified as the parameter direction in the `@Parameter` annotation in the task selection interface. This is a remarkable difference, especially for applications with complex dependencies: while in Taverna the user would have to figure out and draw all the dependencies, which is an error-prone task, Java StarSs would find those dependencies automatically.

- *Dynamicity*: the graph generated by a Java StarSs application is never static, it can vary depending on the input parameters and data, and is subject to the result of control flow statements ('if' conditions, loops, etc.). Taverna also offers some dynamicity (e.g. with implicit loops on lists) but it is more restrictive on what can be drawn, as exemplified in the Hmmpfam version of Figure 4.8.

Although this programmability evaluation has focused on tasks/nodes that correspond to computations executed on grids, Taverna also has some support for service invocation. Services can be added to a Taverna workflow as nodes and they can be connected to other nodes, much like in Java StarSs services can be invoked as regular methods and they can exchange data with other tasks. Moreover, both Java StarSs and Taverna allow to create composite services, i.e. services whose tasks/nodes are other services. The use of services in Java StarSs will be further discussed in Chapter 6.

4.5 Experiments

The experiments in this section will be divided in two series. The first series demonstrate how Java StarSs is able to run applications on large-scale heterogeneous grids, as well as to handle various kinds of Grid middleware. As a complement, a second series of tests will evaluate the performance and some features of Java StarSs in smaller and more restricted environments.

4.5.1 Large-Scale Tests

This first series of tests will show how the tasks of an e-Science application are executed in three different grids with Java StarSs. After describing the application and the testbed, the experiments will be discussed.

4.5.1.1 The Discrete Application

DISCRETE [54] is a package devised to simulate the dynamics of proteins using the Discrete Molecular Dynamics (DMD) methods. In such simulations, the

particles are assumed to move with constant velocity until a collision occurs, conserving the total momentum and energy, which drastically saves computation time compared to standard MD protocols.

The simulation program of DISCRETE receives as input a coordinate and a topology files, which are generated with a setup program also included in the package. The coordinate file provides the position of each atom in the structure, and the topology file contains information about the chemical structure of the molecule and the charge of the atoms. Besides, the simulation program reads a parameter file, which basically specifies three values: EPS (Coulomb interactions), FSOLV (solvation) and FVDW (Van Der Waals terms).

The Discrete application is a sequential Java program that makes use of the DISCRETE package. Starting from a set of protein structures, the objective of Discrete is to find the values of the EPS, FSOLV and FVDW parameters that minimise the overall energy obtained when simulating their molecular dynamics with DISCRETE. Hence, Discrete is an example of a parameter-sweeping application: for each parameter, a fixed number of values within a range is considered and a set of simulations (one per structure) is performed for each combination of these values (configuration). Once all the simulations for a specific configuration have completed, the configuration's score is calculated and later compared to the others in order to find the best one.

The main program of the Discrete application, whose code can be found in Appendix A.3.1, is divided in three phases:

1. For each of the N input protein structures, their corresponding topology and coordinate files are generated. These files are independent of the values of EPS, FSOLV and FVDW.

2. Parameter-sweep simulations: a simulation is executed for each configuration and each structure. These simulations do not depend on each other. The more values evaluated for each parameter, the more accurate will be the solution.

3. Finding the configuration with minimal energy: the execution of each simulation outputs a trajectory and an energy file, which are used to calculate a coefficient for each configuration. The main result of the application is the configuration that minimises that coefficient.

Regarding the tasks of the application, a total of six methods were chosen. The following points describe them, the subindexes indicating the phase to which they belong (please refer to Appendix A.2.2 for the task selection interface):

- $genReceptorLigand_1$: given a structure file, it generates some associated files (receptor and ligand). It is invoked N times (one per structure).

- $dmdSetup_1$: it executes the DMDSetup binary, included in the DISCRETE package, with a structure's receptor and ligand as input; as output, it generates the topology and coordinate files for the structure. It is invoked N times (one per structure).

- *simulate*$_2$: it runs the simulation binary of the DISCRETE suite, given a coordinate file, a topology and a specific configuration (FVDW, FSOLV and EPS values); it returns an average score file. If the number of values considered for EPS, FSOLV and FVDW is S_{EPS}, S_{FSOLV} and S_{FVDW}, respectively, this method is invoked $N \times S_{EPS} \times S_{FSOLV} \times S_{FVDW}$ times.

- *merge*$_2$: it merges two average score files belonging to the same configuration of parameters. It is invoked $(N-1) \times S_{EPS} \times S_{FSOLV} \times S_{FVDW}$ times.

- *evaluate*$_3$: it generates the final coefficient from all the average scores of a configuration. It is invoked once per configuration, i.e. $S_{EPS} \times S_{FSOLV} \times S_{FVDW}$ times.

- *min*$_3$: it receives two coefficient files and outputs the lowest one. It is invoked $(S_{EPS} \times S_{FSOLV} \times S_{FVDW}) - 1$ times.

4.5.1.2 Testbed

The Discrete application was executed with Java StarSs on real large-scale scientific grids. The whole infrastructure used in the tests is depicted in Figure 4.9, and it includes three grids: the Open Science Grid, Ibergrid and a small grid owned by the Barcelona Supercomputing Center.

*** Open Science Grid**

The Open Science Grid (OSG) [44] is a science consortium, funded by the United States Department of Energy and the National Science Foundation, that offers an open Grid cyberinfrastructure to the research and academic communities. OSG federates more than 100 sites around the world, most of them located in the United States, including laboratory, campus, and community facilities. These sites provide guaranteed and opportunistic access to shared computing and storage resources. As of May 2011, OSG comprised a total of around 70,000 cores and 29 Petabytes of disk storage and it provided 1.4 million CPU hours/ day [48].

OSG is used by scientists and researchers to perform data analysis tasks that are too computationally intensive for a single data center or supercomputer. This grid was created to process data coming from the Large Hadron Collider at CERN, and consequently most of its resources are allocated for particle physics; however, it is also used by research teams from disciplines like biology, chemistry, astronomy and geographic information systems.

Each of the OSG sites - clusters, computing farms - is configured to deploy a set of Grid services, like user authorisation, job submission and storage management. Basically, a site is organised in a *Compute Element* (CE), running in a front-end node known as the *gatekeeper*, plus several *worker nodes* (or execution nodes). The CE allows users to run jobs on a site by means of the Globus GRAM (Grid Resource Allocation Manager) [117] interface; at the back-end of

this GRAM gatekeeper, each site features one or more local batch systems - like Condor [174], PBS [43] or LSF [49] - that process a queue of jobs and schedule them on the worker nodes. Besides, the standard CE installation includes a GridFTP server; typically, the files uploaded to this server are accessible from all the nodes of the site via a distributed file system like NFS (Network File System [39]).

*** Ibergrid**

Ibergrid was set up in May 2010 as an umbrella organisation for ES-NGI [56] and INGRID [29] - the Spanish and Portuguese National Grid Initiatives, respectively - in the framework of the European Grid Initiative (EGI), which has the mission of creating and maintaining a pan-European Grid infrastructure.

Ibergrid offers aggregated computing power of more than 24,000 cores and 20 Petabytes of online storage and supports scientists in several fields of research, including high-energy physics, computational chemistry, engineering and nuclear fusion. Ibergrid also dedicates, like the OSG, a significant part of its resources to process data from the LHC. In total, the usage of Ibergrid reached 124 million CPU hours in 2011 [26].

Similarly to OSG, the Ibergrid infrastructure is composed by different sites, each one with a gatekeeper node interfacing to the cluster, a local resource management system (batch) and a set of worker nodes. However, in Ibergrid the middleware installed is gLite [132] and job management is a bit different: instead of submitting the jobs to a given CE directly, the user proceeds by interacting with a *Workload Management Server* (WMS), which acts as a meta-scheduling server. Therefore, matchmaking is performed at a higher level: the WMS interrogates the Information Supermarket (an internal cache of information) to determine the status of computational and storage resources, and the File Catalogue to find the location of any required input files; based on that information, the WMS selects a CE where to execute the job.

*** BSC Grid**

Finally, the Barcelona Supercomputing Center (BSC) Grid [9] is a small cluster located in the BSC premises and formed by five nodes. Three of them have a single-core processor at 3.60GHz, 1 GB of RAM and a local disk of 60 GB. The other two have a quad-core processor at 2.50GHz each core, 4 GB of RAM and a local disk of 260 GB. The cluster does not have any shared file system configured.

The BSC Grid is the only grid of the testbed that supports interactive execution: the user can connect to any of the nodes separately via SSH and launch computations on them. Moreover, files can be transferred to/from the local disk of each node through SSH as well.

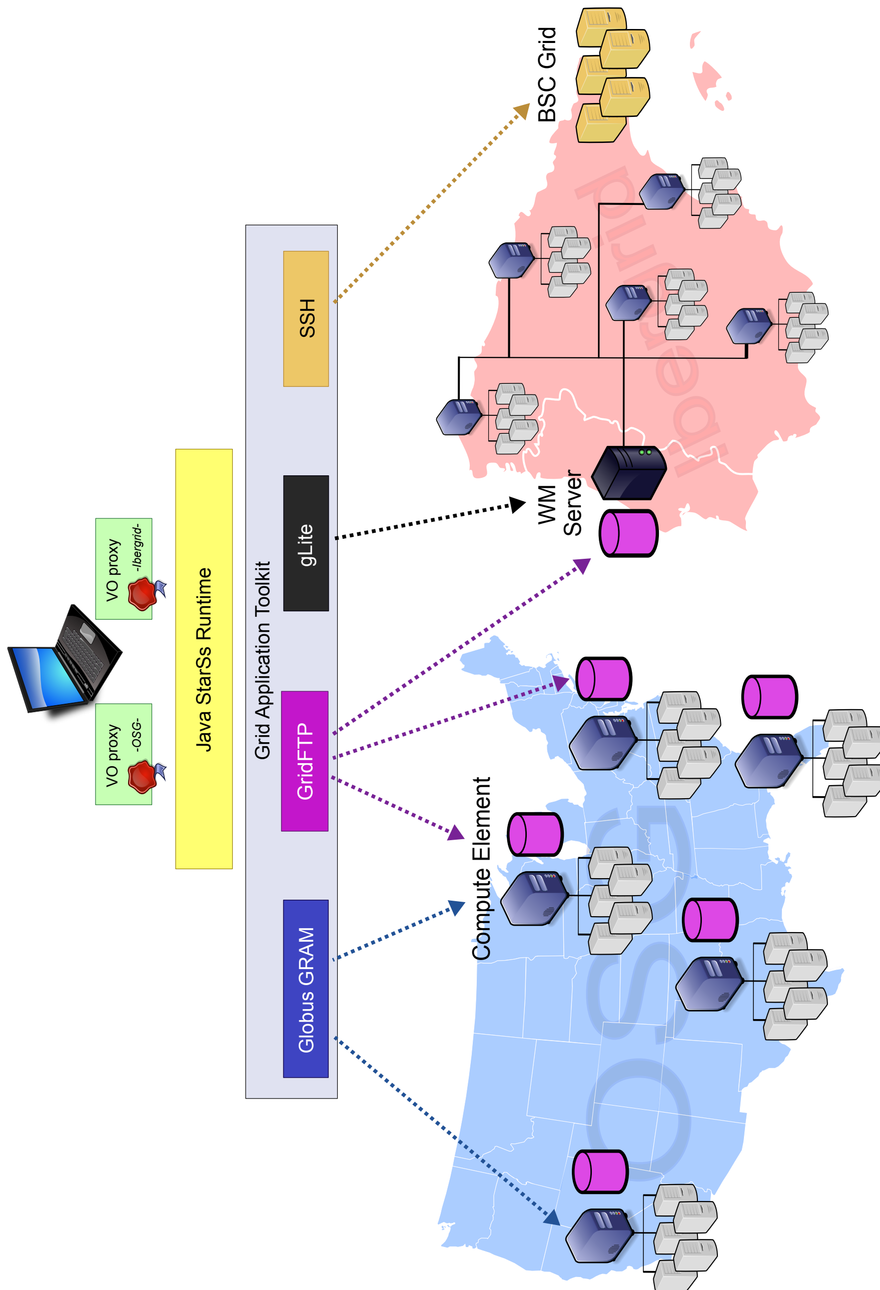


Figure 4.9: Testbed comprising two large-scale scientific grids (Open Science Grid, Ibergrid) and a local BSC-owned grid. The Discrete application, running on a laptop with Java StarSs, interacts with the grids through GAT and its middleware adaptors.

4.5.1.3 Results

*** Configuration and Operation Details**

In order to run the Discrete application in the described testbed, the testing environment was configured as shown in Figure 4.9.

The access point to the Grid was a laptop equipped with a dual-core 2.8 GHz processor and 8 GB RAM. This is different from the traditional procedure of submitting jobs from a User Interface node (UI) of a grid, where the software to interact with that grid is already present. Since the experiments did not target a particular grid but three different ones, and to illustrate how a user can execute a Java StarSs application on the Grid from her own machine, another approach was followed.

The laptop hosted the main program of the application, and therefore it had the Java StarSs runtime and the JavaGAT library installed. Notice that no client middleware had to be installed in the laptop, the GAT adaptors sufficed to interact with all the grids. In addition, prior to the execution, the credentials for each grid were obtained. Putting aside the setup of the SSH keys to access the BSC Grid, OSG and Ibergrid required proxy certificates for authentication, each with a different VO extension. Both proxies were created in a UI node of Ibergrid with the VOMS tools [67] and then placed in the laptop, so that JavaGAT could make use of those credentials when contacting the grids.

Concerning the Grid middleware, the points below list the GAT adaptors and the corresponding grids where they were used. The resources available in each grid were specified in a resources file, along with their capabilities (e.g. associated storage servers).

- *Globus GRAM and OSG*: a total of six OSG sites that support our VO (Engage) were used in the tests, each with its own CE. The gatekeeper of every CE was contacted by means of the Globus GRAM adaptor, used for job submission and monitoring in OSG.

- *gLite and Ibergrid*: the gLite adaptor was used to submit and monitor jobs by connecting to an Ibergrid WMS, which is in charge of selecting the execution site in Ibergrid. Among all the WMS at the disposal of our VO (ICT), the one with most availability was chosen.

- *GridFTP (OSG and Ibergrid)*: the OSG CEs and the Ibergrid WMS offer each a GridFTP server. The GAT GridFTP adaptor was used to transfer files to those servers during execution.

- *SSH and BSC Grid*: two nodes of BSC Grid were used in the tests, being accessed through the GAT SSH adaptors for job submission and file transfer.

Before execution, there was a previous phase of deployment where some required files were installed in the grids; those included, on the one hand, the worker runtime and, on the other, the classes and executables of the application tasks. In OSG, the files to be deployed were copied to the GridFTP server of

each CE, so they could be accessed from the worker nodes. In Ibergrid, the files were transferred to the GridFTP server of the WMS, since the final execution site is not known in advance in this scenario; each time a job is created in Ibergrid, those files are copied by the worker runtime from the GridFTP server to the site where the job will run. Finally, in BSC Grid the files were placed in the local disk of the nodes.

At execution time, the master runtime of Java StarSs sends the Discrete tasks and transfers files to the three grids by means of GAT. In OSG, the input files of each task are first pre-staged to the GridFTP server of the target CE, thus being accessible through the NFS server of that CE too; after that, when the job is created in the CE to execute the task, the worker runtime copies the input files from NFS to the local disk of the target worker node; similarly, the output files are copied from local to NFS at the end of the task, thus being available in the GridFTP server as well. In Ibergrid, the task input files are transferred to the GridFTP server of the WMS; the pre and post-staging of those files to/from the final worker node is taken care by gLite: the WMS chooses the execution site, sends the job to the head node of that site, then the job is locally scheduled and the input files are copied from the GridFTP server to the local disk of the worker node (the process is inverse for the output files). Lastly, the BSC Grid scenario is simpler since the files can be directly transferred to/from the local disk of the final execution node.

In the case of Discrete, all the application input files were initially located in the laptop's disk and then progressively transferred to the execution resources as the application ran; nevertheless, for applications dealing with huge files, the programmer can also refer to those files with a whole URI (i.e. including the resource name) in the application code, so that they are got from that resource.

When scheduling jobs on the grids, the Java StarSs runtime takes into account locality: a task will be assigned, if possible, to a resource that already possesses one or more of the task's input files (in its GridFTP server or local disk). Whenever a resource is freed (a task finishes), the scheduler chooses the task with the best score among the pending ones, the score being the number of task input files in the resource. Note that Ibergrid counts as a single entity for locality, because the final destination of the job is not decided by Java StarSs. If some input file is missing in the chosen resource, such file is replicated to that resource. If the source and destination resources share the same credentials (e.g. two OSG sites) such transfer happens directly between them; otherwise, the file is first copied to the laptop and then to the destination resource.

*** Discussion of the Results**

From the point of view of the application, *all the Grid management discussed above is transparent*. The application deals with its parameters, i.e. number of structures and coefficients. For these experiments, the parameters were the following: $N = 10$ (structures), $S_{EPS} = 3$, $S_{FSOLV} = 3$, $S_{FVDW} = 3$ (i.e. 27 configurations for parameter sweeping). Applying the formulae in Section 4.5.1.1, this leads to a total of 586 tasks - the whole graph can be seen in Appendix A.2.3. Out of those 586, 270 are simulation tasks, which are the most computationally-intensive (between one and two minutes of execution time).

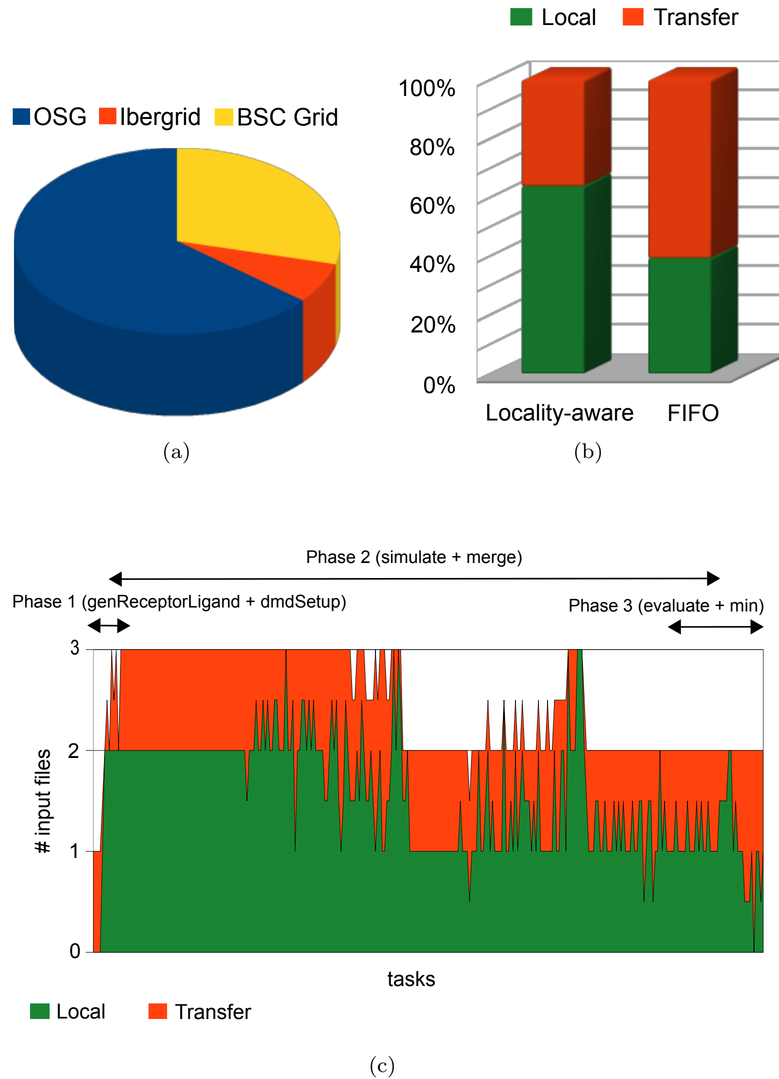


Figure 4.10: Test results for the Discrete application when run with Java StarSs in the Grid testbed: (a) distribution of the Discrete tasks among the three grids; (b) comparison of percentage of transfers between the locality-aware and FIFO scheduling algorithms; (c) evolution of the number of transfers when applying locality-aware scheduling.

Table 4.1: Job submission and file transfer statistics for Discrete.

Grid	Resource	# Job sub.		# File tra.	
		OK	*Failed*	*OK*	*Failed*
OSG	brgw1.renci.org	72	4	102	1
	gridgk01.racf.bnl.gov	43	0	70	1
	rossmann-osg.rcac.purdue.edu	57	14	89	11
	smufarm.physics.smu.edu	69	1	92	1
	stargrid02.rcf.bnl.gov	55	0	90	1
	u2-grid.ccr.buffalo.edu	62	1	96	0
	TOTAL	358	20	539	15
Ibergrid	wms01.ific.uv.es	33	209	58	0
	TOTAL	33	209	58	0
BSC Grid	bscgrid05.bsc.es	122	0	116	0
	bscgrid06.bsc.es	73	0	79	0
	TOTAL	195	0	195	0
	TOTAL	586	229	792	15

Figure 4.10(a) shows how tasks were distributed among the three grids during an execution of Discrete with Java StarSs. The six OSG resources were the ones that consumed more tasks; indeed, among all the OSG sites that support our VO, the ones with most availability were chosen. The two BSC Grid nodes also executed a significant number of tasks because they are directly accessible and therefore those tasks did not suffer from queue waiting times. Ibergrid was the least used of the grids, primarily because of three factors. First, the high job load of Ibergrid resources, which can lead to higher queue times. Second, the internal scheduling policies of the Ibergrid sites, where several sites offer to our VO only opportunistic access to their resources (i.e. when the owners are not using them for other purposes); some other sites reserve a certain number of slots with priority but they are shared by all the Ibergrid VOs. Finally, the errors when submitting jobs to the WMS were frequent, which made tasks go through a (sometimes long) resubmission process.

In that sense, Table 4.1 contains the statistics of errors in job submissions and file transfers for the different grids and a particularly faulty execution of Discrete, in order to demonstrate the fault tolerance mechanisms of the Java StarSs runtime. In general, the OSG sites presented only occasional failures in job submissions and file transfers, which were easily solved with resubmissions and retransfers with no need for task rescheduling. On the contrary, the errors when connecting to the Ibergrid WMS were common, possibly because of the WMS itself or because of a bug in the GAT gLite adaptor; in order to face that issue, several retries were attempted when necessary for a job (6 per job on average), progressively increasing the time between two resubmissions. The most reliable combination of grid/adaptor was BSC Grid/SSH, for which no errors of any kind were registered.

Regarding data locality, Figure 4.10(b) depicts a comparison between two executions of Discrete, one using the locality-aware scheduling algorithm and the other one applying a FIFO (First In First Out) strategy: an incoming task is always assigned to the first available resource on the list, and a freed resource is matched with the first pending task if any. The bars show the percentage of transfers actually performed versus the percentage of locality (the transfer was not necessary because the input file was already on the target execution resource), the total being the number of input files of all tasks. The locality-aware algorithm achieved remarkable results, preventing almost 2 out of every 3 transfers and outperforming FIFO by about 25 %.

As a complement to Figure 4.10(b), Figure 4.10(c) illustrates the number of transfers that could be avoided thanks to locality all along the application execution. The x-axis represents the tasks of Discrete in the order that they are scheduled during the application execution; each point of the axis corresponds to two tasks, so that the number of points is reduced by half and the shape of the plotted lines is smoother. The y-axis reflects the evolution of avoided transfers (Local) and performed transfers (Transfer), each point showing the average of two tasks for both values. In the first phase of the application, the genReceptorLigand tasks require their input files to be transferred from the laptop to the Grid resources, while the successor dmdSetup benefit from full locality because they are scheduled in the same resources as their predecessor tasks, where the corresponding receptor and ligand files are already present. After that, there is an explosion of, first, simulate and, later, merge tasks (wide region of the graph in Appendix A.2.3), for which the runtime can prevent up to three and two transfers, respectively. Finally, the graph gets narrower when the merged scores of the simulations are processed by the evaluate and min tasks, each with two input files subject to locality.

Discrete works with only a few MB of data, but preventing files from being transferred in grids becomes more important as the size of these data increases. Furthermore, when dealing with big files the locality algorithm should take into account not only the number of files but also their size when selecting the destination host of a task. This requires to keep track of the sizes of each file updated/generated in the workers, as well as to send that information to the master runtime for it to make better decisions. Such optimisation was addressed in [156] but it is out of the scope of this thesis. Alternatively, the user can associate a given kind of task that accesses some big input data with a certain resource that is known to host those data, or with a resource that fulfills some other hardware/software requirements of the task.

In that sense, a variant of the tests discussed above intended to demonstrate how to use constraints to force the scheduling of tasks on certain resources. Let us assume that each kind of Discrete task has some hardware/software requirements. Figure 4.11 shows how such requirements were specified by means of the @Constraints annotation, at method level, in the task selection interface (see Chapter 2, Section 2.2.1 for more details). Those requirements need to match the resource capabilities contained in the resources file. In this example, the genReceptorLigand and dmdSetup must be executed in nodes running Scientific

```
public interface DiscreteItf {
    @Constraints(operatingSystem = "Scientific Linux")
    @Method(...)
    void genReceptorLigand(...);

    @Constraints(operatingSystem = "Scientific Linux")
    @Method(...)
    void dmdSetup(...);

    @Constraints(appSoftware = "DISCRETE")
    @Method(...)
    void simulate(...);

    @Constraints(appSoftware = "DISCRETE")
    @Method(...)
    void merge(...);

    @Constraints(memory = 4)
    @Method(...)
    void evaluate(...);

    @Constraints(memory = 4)
    @Method(...)
    void min(...);
}
```

Figure 4.11: Detail of the task constraint specification for the Discrete application. The complete task selection interface can be found in Appendix A.2.2.

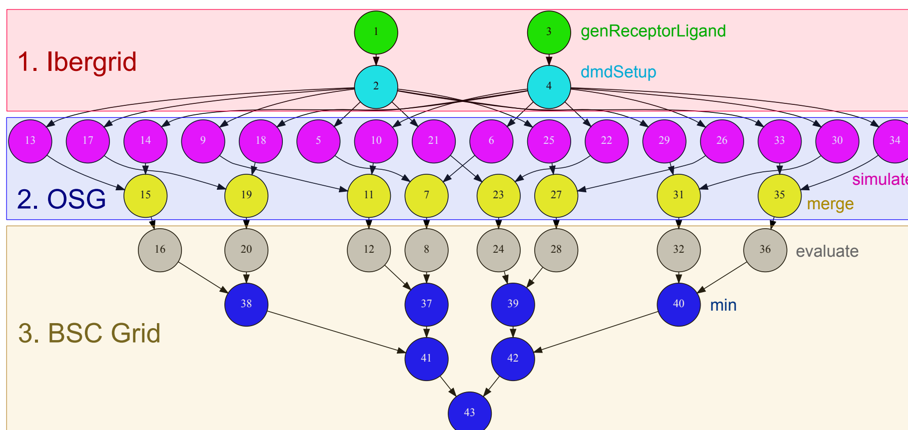


Figure 4.12: Reduced version of the Discrete graph, only for illustrative purposes (the real one is in Appendix A.2.3). The constraints in Figure 4.11 lead to the task scheduling on the three grids represented by this figure.

Linux, which is the operating system installed in some of the grids that process LHC data, such as Ibergrid. Second, simulate and merge are supposed to run in resources where the DISCRETE software is present; here, such capability was assigned only to OSG sites. Finally, evaluate and min have a hardware constraint attached - more precisely, the amount of physical memory - which was only known and specified in the resources file for the BSC Grid nodes.

As a result of the constraints, at execution time the scheduling of tasks on resources was the one depicted in Figure 4.12. This graph is a smaller version (only 2 structures and 8 configurations) just for illustration purposes, the actual graph can be found in Appendix A.2.3. In conclusion, the programmer can use task constraints to make sure that a given group of tasks will be executed in one or more resources that conform to a set of requirements.

4.5.2 Small-Scale Tests

In order to complement Section 4.5.1, this section presents some early experiments that analyse other aspects of the Grid runtime, like component distribution and scheduling techniques. The testbeds in this case are smaller-scale and cluster-like; this represents a more controlled environment that facilitates the evaluation of those aspects.

4.5.2.1 Component Distribution in Nord

A first series of experiments took place in Nord, a cluster of 28 nodes, each node equipped with two single-core PowerPC 970FX processors. A total of 18 nodes were available for the tests, whose objective was to demonstrate the advantages of component distribution when applied to the Java StarSs Grid runtime.

Initially, the tests were intended to execute with JavaGAT over SSH for inter-node communication (job submission and file transfer). Nevertheless, the SSH job submission adaptor of JavaGAT at that time (2008) scaled poorly as the number of simultaneous jobs and worker nodes was increased, causing an overhead that hindered the runtime from distributing tasks quickly enough. For such reason, these tests used a modified Job Manager component that called directly the JSch SSH libraries [34], instead of through the JavaGAT API.

Regarding the applications, two different benchmarks were used. First, *Matmul* multiplies two matrices divided in blocks, which are themselves smaller matrices of doubles. Each of the tasks generated by Matmul multiplies two blocks, stored in files. In the resulting graph, tasks are organised in chains, each calculating the value of one block of the output matrix.

Second, *Mergesort* sorts a list of integers using the merge-sort algorithm. In a first phase, the input list of Mergesort is split recursively into sublists of length 1; then, in a second phase, the sublists are merged back into a sorted list, also in a recursive way. Therefore, there are two kinds of task: the ones that split a list and the ones that merge two lists. Concerning the dependency graph, the number of split tasks that can be run in parallel grows as a power of 2 until reaching a maximum value of $N/2$, N being the length of the original

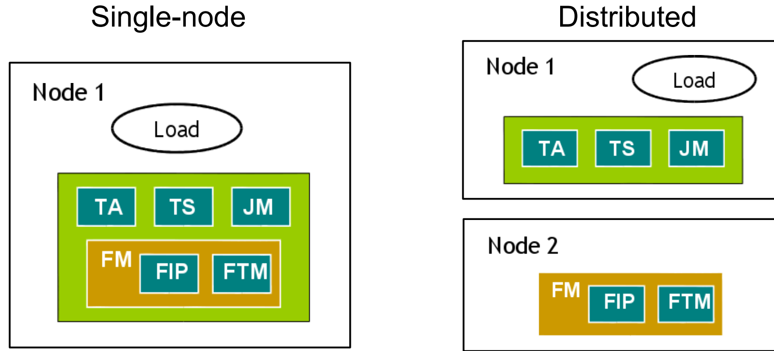


Figure 4.13: Deployments of the Mergesort runtime: Single-node and Distributed.

list, whereas the number of parallel merge tasks decrease in the same ratio. The transition between the two phases causes the maximum stress on the runtime, which has to process a huge number of fine-grained tasks if the input list is long.

In order to show how the distribution of components can contribute to alleviate overload conditions, we launched two applications with Java StarSs from the same node: first, Matmul with input matrices of 32x32 blocks, 8x8 doubles per block; second, Mergesort with an input list of 200 integers. Matmul was started 3 minutes before Mergesort, to make Mergesort begin when Matmul was already on its peak of task load. Furthermore, two configurations were considered: in the first one, the Java StarSs runtimes for Matmul and Mergesort were both completely deployed in the same node, while in the second one the File Manager component of the Mergesort runtime was remotely deployed in a second node, by means of a modified deployment descriptor; Figure 4.13 shows the two deployments of the Mergesort runtime. Concerning the workers, a total of 16 nodes (32 processors) were used.

Table 4.2 contains the average time (in seconds) corresponding to the execution of Mergesort with the two configurations (*Single-node* and *Distributed*). We distinguish three periods: *Start*, including the component deployment, start and initialization, *Task processing*, related to the analysis, scheduling and submission of all the application tasks, and lastly *Finalisation*, comprising the component cleanup and termination.

On the one hand, the remote deployment does not introduce an overhead concerning the start and finalisation of the components, and even the Start period takes less time thanks to the distribution of the component start and initialization. Similarly, the time spent in the Task processing period is smaller when using the distributed configuration; the reason is clear: in the case of the Single-node configuration, the master becomes a bottleneck when having to process a huge number of tasks, especially at the transition between the split and merge phases, while on the contrary the distributed Mergesort runtime divides its load, leading to better results. In total, the Distributed configuration

Table 4.2: Influence of component distribution in Mergesort

Configuration of master	**Period of the execution (in seconds)**		
	Start	**Task processing**	**Finalisation**
Single-node	85	251	8.7
Distributed	64.3	198.3	8.7

outperforms by more than 20% the Single-node one, thus proving the benefits of distributing an overloaded master runtime.

4.5.2.2 Hmmpfam in MareNostrum

A second series of experiments were carried out in the MareNostrum supercomputer, equipped with IBM PowerPC 970MP processors at 2.3 GHz, which are organised in JS21 blades of 2 dual-core processors, 8 GB RAM and 36 GB of local storage. 280 TB of additional shared storage are provided via the General Parallel File System (GPFS) [27]. The interconnection network used was a Gigabit Ethernet.

These tests evaluate the performance of the Grid runtime of Java StarSs when running the Hmmpfam application, described in Section 4.4.3.1.

*** Speedup Measures**

A first kind of experiment measured the speedup and scalability of Java StarSs running HMMPfam. Besides, the same tests were run with a reference parallel implementation of hmmpfam, included in the MPI-HMMER suite [182]. MPI hmmpfam is based on a master-worker paradigm. Each worker must have a copy of the HMM database available either locally or via a network storage. The master distributes to the workers the sequences and the indexes of the HMMs to compute, and finally post-processes the results from all workers.

Concerning the execution parameters, the input files were Superfamily [58] as the HMM database and a set of 5000 sequences produced for benchmarking by researchers from the European Bioinformatics Institute (EBI) [13].

In the case of Java StarSs, the SSH JavaGAT adaptor (updated to the 2010 version) was used to perform job submission and file transfer operations. In every execution of Hmmpfam with Java StarSs, both the database and the sequences file were initially located on the shared storage (GPFS); when starting the execution, the database and/or the sequences file were segmented by Hmmpfam and the fragments were put in the local storage of the master node. From that moment on, the fragments were transferred via SSH from the master to the local storage of a worker or between workers, depending on the scheduling of the tasks, which was locality-aware. The fragments of database or sequences were not stored in GPFS because of performance issues when a number of nodes is accessing the same shared GPFS file. Besides, the I/O-bound nature of hmmpfam would make the problem worse and would prevent the application from scaling. For the same reason, in the case of MPI-HMMER we pre-distributed the database file to the local store of each worker node before the execution.

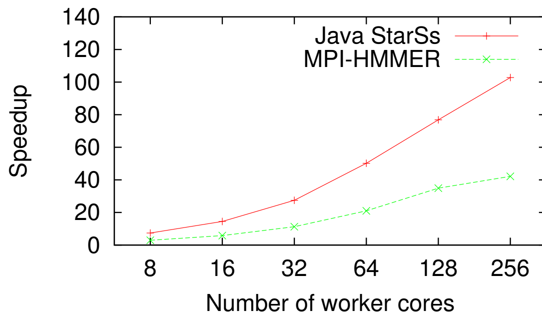


Figure 4.14: Performance comparison for Hmmpfam between Java StarSs and MPI-HMMER.

Regarding the segmentation strategy in Hmmpfam-Java StarSs, the application only produced fragments of the query sequences file, since the database fitted in memory and so there was no real need to segment it. The number of sequence fragments was set to 512, based on previous experiments, to obtain a good tradeoff between the overhead of task processing and the load balancing among resources.

Figure 4.14 compares the performance of Hmmpfam for the Java StarSs and MPI-HMMER versions. The baseline of the speedups is a sequential run of the hmmpfam binary with the aforementioned input parameters.

On the one hand, Figure 4.14 shows the lack of scalability of MPI-HMMER Hmmpfam, which is mainly due to the excessive communication between the master and the workers. This was previously stated in [183] and [126], which propose enhanced versions of the HMMER suite that achieve better performance than the original one. We did not choose any of these works to establish a comparison with Java StarSs because they either modify the original hmmpfam code to introduce optimisations or they target a particular infrastructure. On the contrary, the Grid runtime of Java StarSs relies on standard I/O, makes use of commodity hardware and the hmmpfam code has not been tuned; therefore, we found that the comparison with MPI-HMMER was more adequate.

On the other hand, Hmmpfam with Java StarSs clearly exhibits better scalability than its competitor despite having to transfer the database file, which is about 370 MB, to all the worker nodes during the execution. On the contrary, the times of MPI-HMMER do not include the pre-distribution of the database to all the worker nodes before execution.

In terms of speedup, Java StarSs Hmmpfam still has room for improvement, mainly by increasing the throughput of the master node (tasks processed and submitted per unit of time) but, yet, the results are quite satisfactory, achieving 100x with 256 workers; it is worth noting that, in a revision of MPI-HMMER [183] at that time (2010), the authors only reached a speedup of 80x without modifying the hmmpfam binary to cache the database.

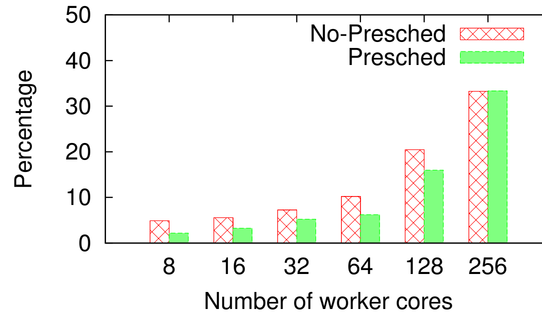


Figure 4.15: Execution of Hmmpfam with Java StarSs. The figure depicts the percentage of Idle+Transferring time in the workers, with respect to the total of Idle+Transferring+Computing, with and without pre-scheduling.

*** Benefits of Pre-Scheduling**

The pre-scheduling technique implemented by the Grid runtime and described in Chapter 3, Section 3.6.1.2 is analysed in this subsection.

During the execution of an application, it can happen that the master becomes idle when all the dependency-free tasks have been scheduled and submitted to their destination host. Later, when a task finishes, the master leaves its idle state to update the dependency graph and possibly send a new task to the freed resource, along with its input files.

Pre-scheduling makes the most of the master inactivity periods, assigning tasks to busy resources and pre-transferring the files that they need to these resources. Thus, the overlapping of computation and communication in the workers is improved.

Figure 4.15 shows how pre-scheduling contributes to shrink the non-computing periods in the workers, for several runs of Hmmpfam with Java StarSs and different numbers of worker processors. During execution, a worker processor can be in three states:

- *Idle* (I): the worker is inactive, that is to say, it is not running any task nor receiving any file.
- *Transferring* (T): the worker is receiving one or more input files for a task.
- *Computing* (C): the worker is running a task.

In Figure 4.15, the percentage of non-computing time in the workers (I+T) is calculated with respect to the total time (I+T+C). The lower is the I+T percentage, the higher is the utilisation of the workers. As expected, the weight of I+T increases along the x-axis due to two factors: first, the more worker cores, the less data locality is achieved and the more transfers are necessary; second, a higher number of worker cores also increases the load of the master, which causes larger idle periods on workers that have finished a task and are

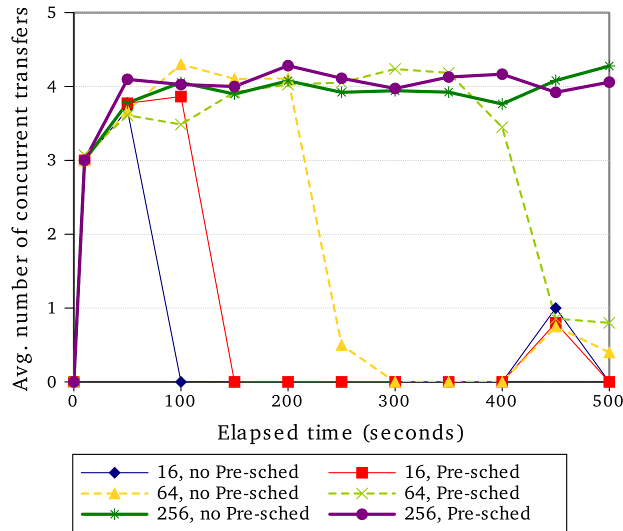


Figure 4.16: Number of concurrent transfers that Java StarSs is performing during the first 500 seconds of Hmmpfam, varying the number of worker cores (16, 64, 256) and applying pre-scheduling or not. Pre-scheduling keeps the master busy (transferring) longer, except in case of overload.

waiting for the next one. However, when pre-scheduling is activated the I+T percentage is smaller. This happens because more transfers are overlapped with computation, and a core that gets free can receive sooner a new (pre-scheduled) task, without having to wait for any transfer at that point. Such statement is not true for the case of 256 workers, when the overload of the master prevents it from applying pre-scheduling: it continuously has newly idle workers to which to transfer files and submit tasks.

Figure 4.16 illustrates how pre-scheduling helps distributing the load of the master more uniformly all along the execution of Hmmpfam. It depicts, for different numbers of worker cores, the average number of concurrent transfers that the master is handling during the first 500 seconds of the execution, both for pre-scheduling and no pre-scheduling. For 16 workers, the pre-schedule line falls about 50 seconds later than the other one, due to the pre-transfers that the master performs. The difference between the two lines is more significant for 64 workers, because the master has more workers to which to transfer input files of pre-scheduled tasks. In the case of 256 workers, however, there is no noticeable difference between the two lines: no pre-scheduling is actually done, again because the overloaded master is never idle.

4.6 Related Work

4.6.1 Grid Programming Models

Apart from Java StarSs, there exist other programming models for applications executed in computational grids [133].

Ninf-G [166] provides an implementation of the GridRPC API, a standard of the Open Grid Forum. GridRPC offers a programming model based on client-server remote procedure calls on the Grid: client programs can call libraries on remote resources using the client API provided by a GridRPC system. Ninf-G uses specific Grid middleware, the Globus Toolkit, to submit the calls from the client (tasks) to the server where the executables reside, whereas Java StarSs can submit tasks using different kinds of Grid middleware. Furthermore, Ninf-G has a more complex programming model than Java StarSs: with Ninf-G, the programmer has to substantially modify the original application code by including the invocations to the GridRPC API. Finally, Java StarSs features a complex mechanism of data dependency analysis for tasks, which Ninf-G lacks.

Satin [175] is a Java-based programming model for the Grid which permits to express divide-and-conquer parallelism; it uses marker interfaces to indicate that certain method invocations need to be considered for potentially parallel (spawned) execution; similarly to Java StarSs, Satin features a bytecode rewriter that transforms such invocations into tasks. Nevertheless, the programmer must explicitly use a synchronisation primitive to wait for the spawned tasks; unlike Satin, Java StarSs takes care of task and data synchronisation automatically. On the other hand, Satin supports shared objects that can be accessed by different tasks, but the programmer must mark the methods invoked on those objects as global (applied to all the object replicas) or local (applied only to the local copy owned by the task); in Java StarSs, data sharing is only achieved through task parameters. Finally, Java StarSs is not restricted to the divide-and-conquer paradigm but targets all applications with potentially concurrent tasks.

OpenWP [98] is a programming and runtime environment that aims to ease the adaptation and execution of already existing applications on grids. For that purpose, OpenWP provides a set of directives, inspired by OpenMP [148], that have to be included in the application code to express parallelism and distribution. These directives allow to run coarse-grain parts of the application (tasks) in parallel on the Grid. Regarding the workflow engine, OpenWP works on top of Condor DAGMan [11], while Java StarSs can access various middleware through JavaGAT. From the programming model point of view, the main difference between Java StarSs and OpenWP is that the latter requires to indicate the dependencies between tasks in the application code, whereas the former finds them automatically at execution time.

ASSIST [76] is a programming environment that makes possible the development of parallel and distributed applications. It offers a coordination language to express parallel programs in two main parts: a module graph which defines how nodes interact using data flow streams, and a set of modules, either sequential or parallel, which actually implement the nodes of the graph; in addition,

a module or a whole graph can be wrapped as a component interoperable with Web Services. ASSIST and Java StarSs have distinct purposes: while the former gives support to high-performance Grid-aware applications, the latter offers a much simpler programming model that is oriented to Grid-unaware applications.

Finally, special mention is deserved by GRID superscalar (GRIDSs) [85], the starting point of the work on Java StarSs. GRIDSs offers a programming model and an execution runtime for Grid-unaware applications. The programming model is also task-based: similarly to Java StarSs, the user is required to provide an IDL (Interface Definition Language) file that selects the tasks and provides the type and direction of their parameters. Besides, the main program of the application must use a set of API methods, e.g. to start/stop the runtime and to wait for some data; in contrast, Java StarSs applications do not need to include any library call. The parameter types of GRIDSs tasks are restricted to files and primitives, while Java StarSs tasks can handle any kind of data used in a Java program. The runtimes of Java StarSs and GRIDSs have similar functionalities; in the case of data dependency analysis, renaming and transfer, Java StarSs extends them to deal with other kinds of data (objects and arrays). The runtime of GRIDSs is programmed in C++ and works on top of the Globus Toolkit and SSH, whereas the Java StarSs runtime is written in Java and can access several kinds of Grid middleware. The work on GRIDSs has been discontinued to be substituted by Java StarSs.

4.6.2 Workflow Managers

With respect to workflow managers, some systems have been proposed to specify the elements of a workflow and the connections between them, either graphically or by means of a high-level workflow description language; in this sense they differ from Java StarSs, where the workflow graph is implicitly defined by a concrete execution of an application and built dynamically at runtime. In addition to the already discussed Taverna [142] (Section 4.4.1), other examples of these systems are P-GRADE, Triana, ASKALON and Pegasus.

P-GRADE [127] is a general-purpose, workflow-oriented, Globus-based Grid portal; it offers a high-level, graphical workflow development system and an execution environment for various grids. Triana [167] is a Problem-Solving Environment (PSE) that permits to describe applications by dragging and dropping their components and connecting them together to build a workflow graph; like in Java StarSs, Triana workflows can access the Grid through the Grid Application Toolkit. ASKALON [145] is an application development and computing environment that makes it possible, through the use of a portal, to create a UML model of a workflow; in a second step, this model is automatically translated to an abstract language that represents the workflow and then given to a set of middleware services for scheduling and reliable execution on the Grid. Pegasus [104] is a workflow management system that takes high-level workflow descriptions (abstract workflows) and automatically maps them to the distributed Grid resources; Pegasus performs execution site selection, manages the input data and provides directives for data transfer and registration.

4.6.3 Component-Based Grid Software

Regarding the CBSE field, some efforts were made to componentise Grid middleware and applications. [93] presents a component-based design of the runtime of the Science Experimental Grid Laboratory (SEGL), a programming environment which allows end-users to program and manage complex, computation-intensive simulation and modeling experiments for science and engineering; this work, though, merely specifies the architecture of a Grid middleware with the CORBA component model [73], whereas Java StarSs also provides a fully-functional implementation and complies with GCM, intended for the Grid.

A good example of a component-based application can be found in [149], where the authors transform an object-oriented distributed numerical solver by applying the features of the Fractal model [97] (predecessor of GCM) with some extensions.

Finally, some approaches to build applications as a set of components have been proposed, for instance the Grid IDE, HOC-SA and CB-PSE; however, only the first one can construct GCM components. The Grid Integrated Development Environment (GIDE) [87] provides an integrated environment to support both the software development process and operation of GCM applications; it is released as plugins for the Eclipse framework [60]. HOC-SA [79] is a programming environment for constructing Grid applications as a composition of Higher-Order Components (HOCs), which implement generic and reusable patterns of parallelism provided as a collection of Grid Services; HOCs can be customized by parameterising them not only with data but also with application-specific code. CB-PSE [181] is a distributed problem-solving environment for scientific computing, which can be used to graphically build applications by connecting software components together; such components can be JavaBeans [57] or CORBA objects that contain sequential or parallel code.

4.7 Summary

This chapter has provided an overview of the first infrastructure contemplated in this dissertation: the Grid. Such infrastructure is mainly characterised by the heterogeneity and geographic distribution of its resources, as well as by the diversity of the middleware that manages those resources. Applications that run on the Grid face the challenge of overcoming that heterogeneity and exploiting the computing power and storage capacity shared by Grid resources. In that sense, the first version of Java StarSs came along to help with the development and execution of Grid applications.

On the one hand, the Java StarSs programming model first focused on files, since they are the Grid's main unit of data; files can be passed as parameters of tasks and accessed from the main program by opening streams on them, like it would happen in any sequential Java application. Regarding tasks, they are preferably coarse-grained in order to compensate the latencies, waiting times and middleware overhead typical of grids.

On the other hand, the Grid runtime of Java StarSs was designed with the Grid characteristics in mind. First, in order to deal with the variety in Grid middleware, the runtime accesses Grid services through a uniform interface with a set of adaptors for several kinds of middleware; this fact is completely transparent to the application programmer, who only needs to provide the necessary credentials for each grid. Second, the structure of the runtime was componentised following the principles of a component model particularly intended for the Grid, thus gaining in reusability, ease and flexibility of deployment in Grid contexts, parallelisation and separation of concerns. Furthermore, the functionalities encompassed by each component were also adapted to the Grid, e.g. by supporting fault-tolerance mechanisms or enforcing data locality in task scheduling.

The evaluation of Grid Java StarSs, both in terms of programmability and performance, was done with applications coming from e-Science, which is the main field where Grid technologies are applied. Such evaluation has highlighted the benefits of using Java StarSs in comparison to well-known alternatives in the field; besides, it has also demonstrated how the Grid runtime can exploit large-scale heterogeneous grids, managed by different middleware and belonging to distinct administrative domains, while abstracting the application from any Grid-related detail.

The last experiments presented in this chapter represent a transition to the Cluster scenario, providing the first performance results of the Grid runtime. Although such results were satisfactory enough for the considered applications, the main conclusion was that there was still room for improvement when executing on clusters. The Grid and Cluster environments have different characteristics, and therefore the design and technologies that are suitable for grids are not necessarily convenient for clusters. For instance, the GAT API and adaptors make less sense in a homogeneous environment like a cluster, and the componentised structure of the runtime introduces some overhead that might not be tolerable for cluster applications.

Therefore, Chapter 5 will explore the changes that were made to Java StarSs in order to tackle the Cluster characteristics and, as a result, improve its performance.

Chapter 5

Cluster

This chapter continues the overview of those parallel distributed infrastructures where Java StarSs has been applied, and it does so by focusing on the *Cluster computing* paradigm. The Java StarSs programming model and runtime, initially designed for the Grid, had to evolve in order to optimise the execution on clusters. The characteristics, technologies and types of application that are particular to Cluster computing motivated such adaptation.

The content of the chapter is organised in the following points: first, a short introduction to the context of Cluster computing and to some basic concepts; second, an explanation of the runtime design decisions driven by the scenario; third, a description of the technologies that influenced the runtime implementation for clusters; fourth, a programmability evaluation of the programming model, comparing it to another approach in the area; fifth, the results of the experiments carried out in clusters; finally, a related work section and a concluding summary.

5.1 Context

5.1.1 Cluster Computing

A cluster can be defined as a type of parallel system that consists of interconnected whole computers and is used as a single, unified computing resource [153]. Clusters appeared decades ago as applications needed more computing power than a sequential computer could provide; instead of improving the speed of a single processor or increasing the amount of memory to meet the demands of applications, cluster computing proposed a alternative solution by connecting multiple processors together and coordinating their computational efforts.

In the early 1990's, the availability of low-price microprocessors and the advances in network technologies contributed to the widespread construction of clusters, which were more cost-effective than specialised proprietary parallel supercomputers [86]. As a consequence, a much broader community could benefit

from powerful computing resources, thus creating new opportunities in sectors like science, industry and commerce.

Nowadays, computer clusters have a wide range of applicability, designs and configurations: from small business clusters with a few nodes built with commodity hardware, to large and expensive supercomputers with hundreds of thousands of cores; from web-based applications to scientific HPC programs. Clusters have already incorporated multi-core processor technologies and are currently exploring solutions like hybrid CPU (Central Processing Unit) - GPU (Graphics Processing Unit) platforms.

5.1.2 Cluster versus Grid

Grids usually integrate clusters as building blocks, as seen in the Grid testbeds of Chapter 4. Both clusters and grids emerged to meet the growing demands of applications by interconnecting resources, but they differ in several aspects, summarised in the following points:

1. Area and size: clusters occupy a small, restricted and single-owned area and they normally gather less resources than grids. Cluster resources communicate through Local Area Networks (LAN), oppositely to the WAN links and geographic distribution of Grid resources across multiple administrative domains.

2. Network: cluster nodes are tightly-coupled and interconnected by dedicated *fast networks*, in some cases featuring very low-latency and high-bandwidth, while grids are typically built on top of slow links.

3. Resources: clusters are usually an aggregation of the same or similar type of machines running the same operating system. This *homogeneity* contrasts with the variety that characterises Grid resources.

4. Applications: an important kind of cluster applications are those that require frequent communication between nodes, mostly implemented with the Message Passing Interface (MPI) [144]. The other extreme corresponds to those applications mainly composed of independent computations that need little or no communication, close to Grid computing where data transfers are more expensive.

5. Data: in addition to files, which are the Grid's main unit of data, many cluster applications work with *memory data* structures that are allocated in the nodes involved in the computation, and eventually sent to other nodes through the network.

6. Granularity: applications that run on grids need to have enough granularity to compensate for the waiting times, middleware overhead and latencies of the Grid. In cluster environments, applications and the computations they encompass are normally more *fine-grained*.

7. Management: unlike the Grid, which has a more distributed nature, Cluster computing relies on a centralised management that makes the nodes available to users as orchestrated shared servers.

5.1.3 Productivity in Cluster Programming: APGAS

In the advent of multi-core processors, next-generation clusters are increasing not only in size but also in complexity. In such a scenario, programming *productivity* - understood as a tradeoff between programmability and performance - is becoming crucial for software developers. Parallel languages and programming models need to provide simple means for developing applications that can run on parallel systems without sacrificing performance.

MPI has dominated so far the programming of HPC applications for infrastructures with distributed memory but, arguably, parallelising an application in MPI requires a considerable effort and expertise. Some of the duties of the MPI developer include manually fragmenting the application data and explicitly managing the communication (sends and receives) between processes.

In response to that fact, in the last years the research community has initiated different projects to create a suitable and robust programming model for distributed-memory platforms like clusters. One of such approaches is that of the Partitioned Global Address Space (PGAS) languages, which came along in order to address the programming-productivity wall. PGAS languages, such as UPC [110], Co-Array Fortran [146] and Titanium [185], extend pre-existing languages (C, Fortran and Java, respectively) with constructs to express parallelism and data distribution. These languages provide a simpler, shared-memory-like programming model, where the address space is partitioned and the programmer has control over the data layout. Besides, they have a strong sense of ownership and locality: each variable is stored in a particular memory segment and, although tasks can access any lexically visible variable, local variables are cheaper to access than remote ones.

Nevertheless, PGAS languages focus on the Single Program Multiple Data (SPMD) threading model and lack support for asynchrony. Therefore, several research groups started to investigate about asynchronous computation in the PGAS model [113], and the concept of APGAS (Asynchronous PGAS) appeared. APGAS languages, such as X10 [101] or Chapel [100], follow the PGAS memory model but they also provide mechanisms for asynchronous execution: they allow to create an activity and return immediately, an activity meaning a computational unit that can run in parallel with the main program. The basis of the APGAS communication model is the Active Message (AM) paradigm [180]. In short, an AM is a message with a header containing the address of a user-space handler to be executed upon message arrival at the receiver, with the contents of the message passed as an argument to the handler; the mission of the AM handler is to extract the data from the network and integrate it into the ongoing computation with a small amount of work.

5.2 Runtime Design

The differences between Cluster and Grid computing enumerated in Section 5.1.2 motivated a change in the design and implementation of the runtime. The modifications mainly affected the following aspects of the Grid flavour of Java StarSs:

- Componentised structure: even if the componentised runtime had interesting properties like reusability and deployability, the ProActive implementation of GCM introduced a considerable overhead primarily due to an inefficient communication protocol between components. Hence, while maintaining the functionalities described in Chapter 3, the runtime was re-designed with *performance* and *scalability* in mind.

- Underlying communication layer: the Grid API and the middleware adaptors were not adequate for execution on a homogeneous environment like a cluster. Instead, the runtime was built on top of a fast communication system capable of exploiting *high-speed networks*.

- New data types: the runtime was extended to support memory structures as task parameters subject to dependencies, as well as to watch their access from the main program. This chapter will focus on *arrays*, whereas Chapter 6 will discuss the use of objects.

- New execution and data model, with persistent workers and data exchange between any pair of nodes.

The next subsections will further explain the modifications made to the Java StarSs runtime design in order to adapt it to Cluster computing.

5.2.1 Java StarSs and APGAS

The APGAS communication system efficiently supports asynchronous execution and data transfers. APGAS is based on a one-sided communication model (by means of AMs), as opposed to the (mostly) two-sided communication pattern in MPI. The one-sided model has proven to achieve better scalability for large-scale clusters, mainly due to its better overlapping of communication and computation and to the avoidance of the inherent bottlenecks of two-sided models (like message matching and preserving ordering semantics [88, 111]). IBM (International Business Machines [30]) developed its own implementation of an APGAS runtime, a fast and portable communication layer that has shown its performance for languages like UPC and X10 [119].

In that sense, it was explored how APGAS could help Java StarSs achieve better performance and scalability. Java StarSs and APGAS share the same computational model, which consists in spawning asynchronous computations as the main program executes, but it was questionable which characteristics of APGAS would be beneficial for Java StarSs. There were basically two APGAS properties of our concern: (i) a partitioned global address space, where every thread is able to locate (address/access) shared data and, at the same time, the

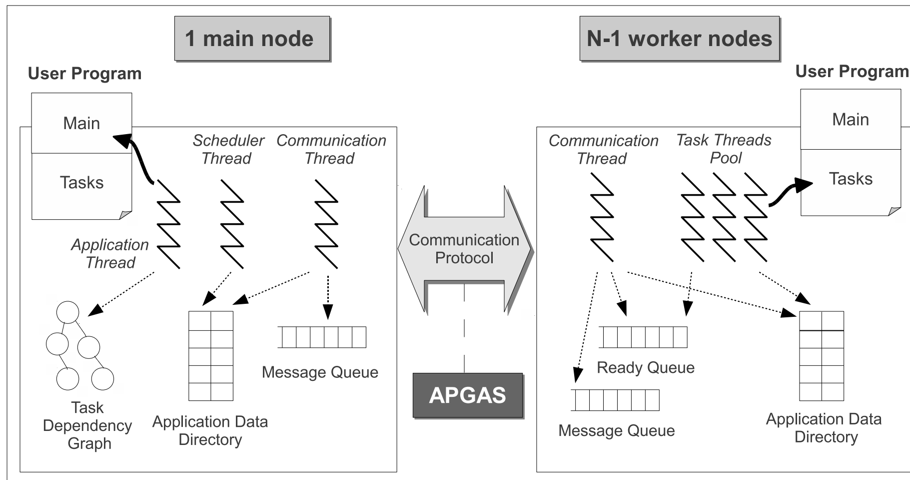


Figure 5.1: Design of Java StarSs on top of APGAS.

concept of affinity allows to exploit data locality; (ii) a communication layer based on one-sided communications.

In the re-design of Java StarSs, only (ii) was taken into account, i.e. APGAS was exclusively used as an underlying layer to handle inter-node communications, following the AM paradigm. With respect to (i), a partitioned global address space permits to increase the address space by adding up every node's memory; nevertheless, the programmer is responsible for specifying the data distribution among nodes and for exploiting data locality. In that regard, it was decided to preserve the simplicity of the Java StarSs model, only requiring the user to select the tasks, while the actual parallel execution and data placement is kept transparent. Indeed, exposing data distribution to the user would lead to a more complicated programming model, and therefore this option was discarded. Instead, our approach is to automatically distribute the application data according to the computation needs, but still being able to use the whole memory of the cluster. The data model in the Cluster design of Java StarSs will be further explained in Section 5.2.5.

Next, the design of the Cluster flavour of Java StarSs will be described, as well as the interaction between Java StarSs and APGAS, in terms of structure of the program executed in all the nodes, the architecture and models of execution and data in Java StarSs. Figure 5.1 illustrates this description.

5.2.2 Runtime Structure

The Java StarSs runtime for clusters is internally implemented as an SPMD program, i.e. all nodes hold a copy of the whole program code. Thus, at execution time, every node starts running the same code, summarized in Figure 5.2.

```
main () {
    initialisation_and_AM_registration();
    barrier();
    if (is_main(here)) {
        application_execution();
    }
    // Workers go directly here and wait for AMs
    barrier();
    cleanup();
    barrier();
}
```

Figure 5.2: Pseudo-code representing the skeleton of the Java StarSs runtime that is run in all nodes. Essentially, the main node executes the main program of the application and the worker nodes wait to respond to incoming AMs.

Every node starts executing the same main method and, after a common stage of initialisation, the behaviour depends on the role of the node, i.e. main node or worker node:

- *Main node manages Task Generation*: the main node runs the actual main program of the application, leading to the generation of tasks and their submission to the workers for execution. It encompasses the main functionalities corresponding to the master runtime, like dependency analysis or task scheduling.
- *Worker nodes manage Task Execution*: the worker nodes spawn threads that wait to respond to incoming messages and execute tasks.

5.2.3 Communication Protocol

The Java StarSs communication protocol for clusters features three kinds of messages:

1. *Task submission* (main node $\rightarrow$ worker): this AM submits a task for execution. It contains the identifier of the method that needs to be executed, plus the list of its parameters and the necessary information for locating them.
2. *Task completion* (main node $\leftarrow$ worker): this AM notifies task finalisation. It triggers an update of the task dependency graph.
3. *Get requests* (main node $\leftrightarrow$ worker, worker $\leftrightarrow$ worker): workers can exchange data, bypassing the main node, to request the input data required by a task. The main node knows the location of the data and such information is included in the Task submission message. If the worker in charge of the task is missing some of the data, it uses the location information to request them directly to the node where they reside (be it the main node or another worker) by means of a Get request message.

Both in the main node and in the workers there is a *Communication thread* that periodically polls the network for incoming messages, queues them in a *Message queue* and then processes this queue. For Task submission messages, such thread uses the information about the location of the task input data to send the appropriate Get requests to obtain those data from other nodes, if necessary. Once all its data are available at the execution node, the task is queued in the *Ready queue*, which is consumed by a pool of *Task threads* that run the tasks. When a task finishes, a Task completion message is sent to the main node to notify that fact; the message is processed by the Communication thread of that node, leading to an update of the task dependency graph.

5.2.4 Execution Model

The main difference between the execution model in the Grid and Cluster scenarios is the persistence of the workers: while in the Grid workers are necessarily transient (Chapter 4, Section 4.2.3), the Cluster runtime features *persistent workers*.

As seen in Section 5.2.2, the Cluster runtime is internally implemented as an SPMD program: on startup, a process is launched at each node and remains there for the entire lifetime of the application. This is possible because all the nodes involved in the computation are known from the start and directly accessible.

One of the benefits of persistency is that workers can keep in memory those data structures passed as task parameters for later use, as discussed next in Section 5.2.5.

5.2.5 Data Model

5.2.5.1 Data Layout

The data created in memory by the main program of the application initially resides in the main node. Furthermore, Java StarSs supports data-allocating tasks: when a task creates and returns some data, such data is allocated directly in the worker node that runs the task.

The possibility of *allocating data by means of a task* prevents the application from being limited to the memory of the main node, which would represent a severe scalability impediment; instead, the total amount of memory is extended to that of all the nodes involved in the computation. Besides, the fact of allocating data in tasks frees the main node from having to transfer those data to the workers.

On the other hand, as explained in Section 5.2.1, the Java StarSs programming model does not permit to explicitly define data distributions. However, it does provide a mechanism to uniformly allocate data among the worker nodes: *initialisation tasks*, exemplified in Section 3.6.1 of Chapter 3, are scheduled in a round-robin manner across the available worker Task threads.

5.2.5.2 Data Transfer

Differently from the Grid case, where the master runtime was always responsible for initiating the transfer between two resources, the Cluster runtime supports data transfers between two worker nodes without the intervention of the main node. Workers extract the information about parameter location from the Task submission message; when a location corresponds to another worker, that worker is contacted directly, *bypassing the main node*. Hence, the main node is freed from processing every single transfer, which contributes to increase scalability.

5.2.5.3 Data Reuse and Locality

Every node in a Cluster Java StarSs execution maintains a structure called *Application Data Directory (ADD)* to manage its task in-memory data. The worker nodes store in the ADD: (i) the task input data transferred from other nodes, (ii) the new versions of data updated by tasks and (iii) the data returned by tasks. Similarly, when some data updated by a task is later accessed by the main program, those data are transferred to the main node and added to its ADD; eventually the main node may update the data and generate a new version which is also stored.

The ADD allows to store renamings for later reuse, thus preventing unnecessary transfers of the same data in the future. As a matter of fact, it is a key structure in the realisation of locality-aware scheduling.

5.3 Relevant Technologies

This section presents the Cluster technologies that were used to implement the runtime design seen in Section 5.2.

5.3.1 IBM APGAS Runtime

The IBM APGAS runtime provides a fast and portable communication system to efficiently exchange messages between the nodes of a cluster. By means of the APGAS API, a given node can start a one-sided communication with another node by sending an active message to the latter. Furthermore, the IBM APGAS runtime is able to exploit high-performance networks and has been implemented on a number of platforms, e.g. Myrinet [112], DCMF for BlueGene [130] and LAPI [124].

As depicted in Figure 5.3, the cluster flavour of the Java StarSs runtime was built on top of the IBM APGAS runtime and its various network adaptors. Java StarSs invokes APGAS through Java bindings that make use of the Java Native Interface [134]; these bindings offer a Java API for calling the actual APGAS runtime API (written in C). The IBM APGAS runtime is used as a communication layer: the APGAS AMs implement the communications between the main node and the workers or between workers, which have been detailed in Section 5.2.3.

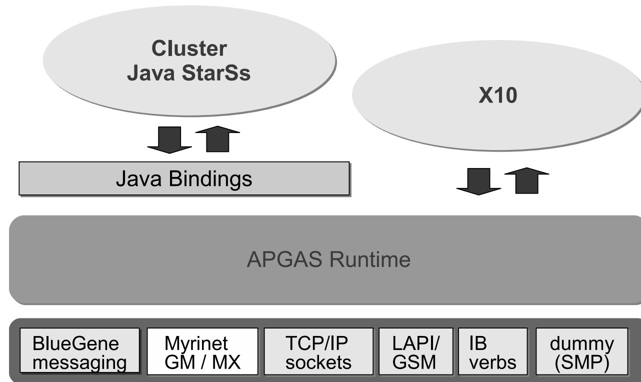


Figure 5.3: Cluster Java StarSs architecture: Java StarSs runtime on top of the APGAS runtime, invoking the latter through Java bindings. X10 shares the same underlying APGAS layer as Java StarSs.

5.4 Programmability Evaluation

This section aims to study the development expressiveness of Java StarSs in the Cluster scenario by comparing it to the X10 language [101]. For that purpose, two linear-algebra benchmarks (Matrix Multiplication and Sparse LU factorisation), as well as the K-means clustering application, were implemented in both languages/programming models.

The next subsections explain the most relevant characteristics of X10 and the chosen applications for their Java StarSs and X10 versions, and finally the programmability of both approaches is discussed.

5.4.1 The X10 Programming Language

X10 [101] is an object-oriented programming language designed by IBM for high-productivity programming of Cluster computing systems.

As in Java StarSs, the X10 execution model is based on spawning asynchronous computations as the application runs. Nevertheless, unlike Java StarSs, X10 provides the programmer with means for decomposing the application's data across a partitioned global address space and for orchestrating the flow of computation through the system; concretely, such means consist in high-level programming language constructs for describing data distribution, creating asynchronous tasks and synchronising them.

This subsection will present a brief summary of the X10 language, focusing on the core features that are used in the applications considered in this programmability analysis.

5.4.1.1 Places and Activities

A central concept in X10 is that of a *place*, which is intended to map to a data-coherent unit in a system, i.e. a node in a cluster. Thus, cluster-level parallelism can be exploited in X10 by creating multiple places. A place acts as a container of both data and asynchronous computations, called *activities*, which enable node-level concurrency.

In an X10 program, place 0 starts executing the main method, from which activities can be spawned either to the same place or to other places. The statement form of an activity is `async (P) S`, where `S` is a statement and `P` is a place expression; such a construct asynchronously creates an activity at the place designated by `P` to execute `S`. Throughout its lifetime, an activity executes at the same place, and has direct access only to data stored at that place; however, an activity can recursively launch additional activities at places of its choosing.

5.4.1.2 Synchronisation

X10 also offers means to synchronise activities. In order to enforce the global termination of a set of activities, they can be enclosed in a `finish` block. Such construct acts as a barrier: it is guaranteed that `finish S` will not complete until all the activities (possibly recursively) generated by `S` have terminated.

5.4.1.3 Data Distribution

X10 is a member of the PGAS (partitioned global address space) languages and, as such, it permits to partition and distribute data across different places, each place being the host and owner of a fragment of those data. Typically, in order to distribute an X10 aggregate object (array), the programmer proceeds in two steps: first, specifying the *region* of the array, i.e. the set of indices for which the array has values; second, determining a *distribution* mapping from indices in the region to places. X10 provides some pre-defined distributions, for instance, to divide the coordinates in one axis in blocks or to assign in a cyclic way coordinates to places.

Since an X10 distributed array belongs to a global address space, any place can have a reference to any element of the array. However, a given element can only be accessed on the node where it resides, more precisely by launching an activity to that place. Reading an array element from a node other than its owner requires an explicit transfer using a copy method of the X10 API. Only constant (immutable) data can be accessed transparently from any place.

5.4.2 Application Description

This subsection describes the linear-algebra benchmarks and the K-means application. Concerning the benchmarks, all of them are programmed using data blocks, i.e. they operate on matrices which are divided in blocks. This kind of algorithms decompose a problem into smaller problems, and they map easily

```
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
        for (int k = 0; k < N; k++)
            multiply(A[i][k], B[k][j], C[i][j]);
```

Figure 5.4: Main algorithm of the matrix multiplication application in Java StarSs. The method multiply multiplies two input blocks of matrices A and B and accumulates the result in an in-out block of matrix C.

into tasks that access parts of the data. The division of matrices in blocks that fit into the cache is a common practice to optimise serial codes for a particular architecture, the block size being chosen depending on the cache size.

In both Java StarSs and X10 the blocks were defined as flat 1D arrays. In Java, this ensures that each block is allocated contiguously, which prevents extra-copies when transferring data through the APGAS runtime.

5.4.2.1 Matrix Multiplication

The main algorithm of the matrix multiplication in Java StarSs (Figure 5.4) multiplies two input matrices A and B and stores the result in C. Each matrix is divided in $N \times N$ blocks of $M \times M$ doubles. The multiplication of two blocks is done by a task (multiply method) with a simple three-nested-loop implementation. When executed with Java StarSs, the program generates N^3 tasks arranged as N^2 chains of N tasks in the dependency graph.

On the other hand, before the code in Figure 5.4 is executed, the blocks of matrices A, B and C are allocated by means of initialisation tasks. Figure 3.4 in Chapter 3 exemplifies how such initialisation is done for the A matrix, the code for the other two matrices being equivalent; only a slight modification to that code was performed in order to flatten the matrix blocks, i.e. to turn them into 1D arrays instead of 2D.

Concerning the X10 implementation, the same algorithm was ported to X10 utilising its parallelism and data distribution constructs. Regarding the data distribution of the matrices, two different configurations were considered:

- C distributed, A and B replicated everywhere: in this configuration, each place (node) is assigned a whole replica of the input matrices A and B and a part of the output matrix C. Figure 5.5(a) shows the code that performs such distribution. Line 1 defines the region of a matrix of $N \times N$ points, each point being a block. Line 2 specifies the distribution of the points across the places, called 'Block distribution' in X10; more precisely, it divides the coordinates along the 0th axis (the rows) in as many parts as there are places[1] and assigns successive parts to successive places (see Figure 5.7(a)). Lines 3-4 do the actual creation of the output matrix following the previously defined distribution; the second parameter of method make is the initialisation function of the blocks that allocates, for each point, a

[1]Here we refer to X10 blocks as 'parts' to prevent confusion with the matrix blocks (the points), even though X10 uses the term 'Block distribution' for this kind of data partitioning.

```
1  val matrix_region = [0,0]..[N-1,N-1];
2  val matrix_dist = Dist.makeBlock(matrix_region, 0);

3  val C = DistArray.make[Rail[Double]](matrix_dist,
4              (p:Point)=>Rail.make[Double](M*M, (Int)=>0 as Double));

5  val A = PlaceLocalHandle.make[ValRail[ValRail[Double]]](
6              Dist.makeUnique(),
7              ()=>ValRail.make[ValRail[Double]](
8                  N*N,
9                  (Int)=>ValRail.make[Double](M*M, (Int)=>2 as Double)));

10  val B = PlaceLocalHandle.make[ValRail[ValRail[Double]]](
11              Dist.makeUnique(),
12              ()=>ValRail.make[ValRail[Double]](
13                  N*N,
14                  (Int)=>ValRail.make[Double](M*M, (Int)=>2 as Double)));
```

(a)

```
1  finish {
2      for (var i:Int = 0; i < N; i++) {
3          for (var j:Int = 0; j < N; j++) {
4              val i_copy = i;
5              val j_copy = j;
6              async (C.dist(i,j)) {
7                  val pij = Point.make(i_copy, j_copy);
8                  for (var k:Int = 0; k < N; k++) {
9                      val k_copy = k;
10                     finish async multiply(A()(i_copy*N + k_copy),
11                                           B()(k_copy*N + j_copy),
12                                           C(pij) as Rail[Double]!);
13                 }
14             }
15         }
16     }
17 }
```

(b)

Figure 5.5: Implementation in X10 of the matrix multiplication benchmark. (a) contains the creation, initialisation and distribution of the three matrices A, B and C involved in the computation. (b) shows the main algorithm.

1-dimensional array (Rail in X10) of doubles filled with zeroes. After that, lines 5-14 create and initialise the two input matrices: the PlaceLocalHandle class will allocate a whole copy of the two matrices at each place; the blocks are again 1D arrays (ValRail = Rail of immutable elements).

- A, B and C distributed: in this case, the three matrices are split across the places. Two partitionings are considered, both of the type 'Block distribution': one along the 0th axis (Figure 5.6(a), line 2), applied to matrices C and A (lines 4-7), and another one along the columns (line 3, example in Figure 5.7(b)), applied to matrix B (lines 8-9).

```
1 val matrix_region = [0,0]..[N-1,N-1];
2 val matrix_dist_0 = Dist.makeBlock(matrix_region, 0);
3 val matrix_dist_1 = Dist.makeBlock(matrix_region, 1);
4 val C = DistArray.make[Rail[Double]](matrix_dist_0,
5               (p:Point)=>Rail.make[Double](M*M, (Int)=>0 as Double));
6 val A = DistArray.make[Rail[Double]](matrix_dist_0,
7               (p:Point)=>Rail.make[Double](M*M, (Int)=>2 as Double));
8 val B = DistArray.make[Rail[Double]](matrix_dist_1,
9               (p:Point)=>Rail.make[Double](M*M, (Int)=>2 as Double));
```

(a)

```
1 val size = M*M;
2 val block:Rail[Double]! = Rail.make[Double](size);
3 val remote_ref = at (A.dist(p)) A(p);
4 block.copyFrom(0, A.dist(p), ()=>Pair[Rail[Double],Int](remote_ref, 0), size);
```

(b)

Figure 5.6: A second implementation of the X10 matrix multiplication. In this version, the three matrices created in (a) are distributed. The main algorithm is not shown since it is equivalent to the one in Figure 5.5(b). The fact of distributing matrices A and B makes necessary to add some code, depicted in (b), to the activity method multiply for explicitly transferring blocks.

The main algorithm, like in the Java StarSs version, is implemented with three nested loops and operates block by block as well. The code is shown in Figure 5.5(b), being equivalent no matter what distribution is chosen. An outermost finish (lines 1-17) guarantees synchronisation for all the activities inside it. A first level of N^2 activities are spawned from place 0 (running the main program) to the places owning each block of C (line 6), since all the updates of a given block must happen at the place where it resides. The i and j variables must be copied into constants (lines 4-5, declared with the val keyword) before invoking async, for them to be automatically transferred to the place where the activity will run. A second level of N activities per C block are locally launched on the place owning each C block (lines 10-12), which perform the actual multiplication of two blocks. The finish construct right before the async ensures that the activities are executed as a sequential chain of computations.

When distributing all the matrices, the subroutine multiply (called from lines 10-12) needs to include some code to explicitly transfer the input blocks from A and B if they are not resident in the place owning the C block. Figure 5.6(b) shows such code, where a reference to the remote point - a block - is obtained (line 3) before it is copied to a local Rail (line 4). Since X10 does not internally cache data copied from other places like Java StarSs does, these transfers can severely affect the performance of the application, as will be seen in Section 5.5.2.

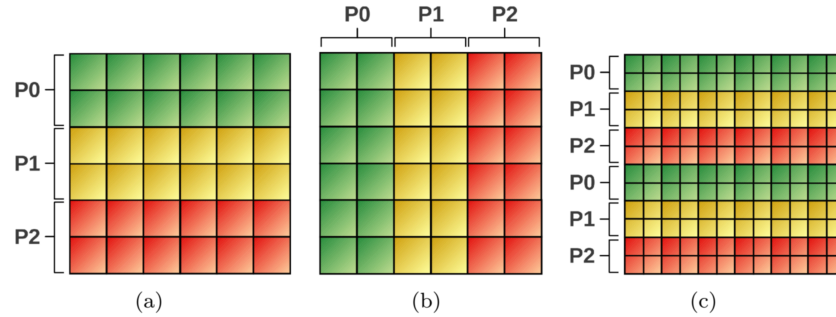


Figure 5.7: X10 matrix distributions used in the tested benchmarks: (a) Block distribution along the 0th axis, (b) Block distribution along the 1st axis, (c) Block Cyclic distribution along the 0th axis with a block size of two rows. In the benchmarks, each cell of a distributed matrix is itself a sub-matrix (i.e. a block of the benchmark).

5.4.2.2 Sparse LU

The Sparse LU kernel computes an LU matrix factorisation on a sparse blocked matrix. The matrix size (number of blocks) and the block size are parameters of the application. As the algorithm progresses, the area of the matrix that is accessed is smaller; concretely, at each iteration, the 0th row and column of the current matrix are discarded. On the other hand, due to the sparseness of the matrix, some of its blocks might not be allocated and, therefore, no work is generated for them.

The Java StarSs version produces several types of task with different granularity and numerous dependencies between them. In a first phase, a group of initialisation tasks sparsely allocate blocks of the matrix. The decision of whether or not to create a block is made according to a certain criteria, and the final placement of the blocks in nodes follows the same pattern as in Figure 3.4(c) in Chapter 3 except that only some of the blocks are allocated. After the initialisation, the actual computation starts. Figure 5.8 depicts both the main algorithm (a) and the graph generated for a matrix size of 5x5 blocks (b). When running the code in (a), the invocations to the methods selected as tasks (in italics: lu0, fwd, bdiv and bmod) are replaced by the asynchronous creation of tasks, which are dynamically added to the graph. This is an example of how Java StarSs is able to deal with an application with complex dependencies, automatically detecting them and trying to exploit the parallelism of the graph as much as possible, while the user programs in a totally sequential fashion.

The X10 implementation is analogous to the Java StarSs one, but they differ in three main aspects: first, the matrix to be factorised is defined as a distributed array; second, there is the need to insert explicit data transfers; third, the synchronisation between activities must be managed by the programmer.

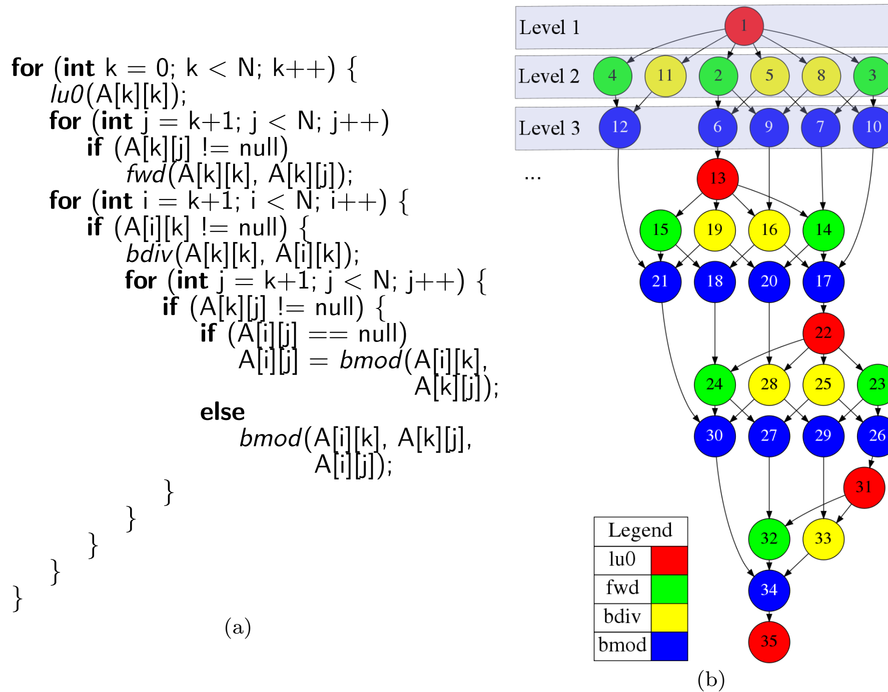


Figure 5.8: (a) Main algorithm of the Sparse LU benchmark for Java StarSs and (b) the corresponding task dependency graph generated for an input matrix of 5x5 blocks. Different node colours in (b) represent different task methods and the number in each node is the generation order. Also in (b), the three highlighted task levels correspond to the three different finish blocks in the X10 implementation.

Concerning the first point, two distributions were considered:

- Block distribution along the 0th axis (Figure 5.7(a)).
- Block Cyclic distribution along the 0th axis (Figure 5.7(c)): this helps alleviate the load unbalance problem of the previous distribution, where the places owning the first rows of the matrix soon get starved since their data is not accessed anymore.

With respect to data copies, the distributed matrix is both read and written by the application: a block written on its owner node might be read later by another node to update a block the latter owns. Since X10 does not handle such accesses transparently, some code to explicitly transfer blocks like the one in Figure 5.6(b) had to be added. In particular, the `fwd` and `bdiv` methods read one block and `bmod` receives two input blocks; inside those methods, such input blocks must be transferred before the computation can begin.

Regarding synchronisation, for every iteration of the main loop three finish blocks were defined, each enclosing a group of activities. Every X10 activity spawned, like in the Java StarSs tasks, runs one of the lu0, fwd, bdiv and bmod methods. The three finish blocks correspond to the task levels in Figure 5.8(b); in that example, the first level has one lu0 task, the second one has six tasks (three fwd, three bdiv), the third one has five bmod, etc. In X10, the activities in each of the levels follow a fork-join pattern: they can run in parallel, but there is a global barrier at the end of the level. Although this means there is less concurrency in the X10 version, trying to program in X10 a synchronisation as fine-grained as the represented by a Java StarSs graph would be unreasonably hard.

5.4.2.3 K-means

K-means clustering is a method of cluster analysis that aims to partition n points into k clusters in which each point belongs to the cluster with the nearest mean. It follows an iterative refinement strategy to find the centers of natural clusters in the data.

The Java StarSs version of K-means first generates the input points by means of initialisation tasks. For parallelism purposes, the points are split in a number of fragments received as parameter, each fragment being created by an initialisation task and filled with random points. Thus, the fragments are allocated in the worker nodes in a round-robin manner. After the initialisation, the algorithm goes through a set of iterations. In every iteration, a computation task is created for each fragment; then, there is a reduction phase where the results of each computation are accumulated two at a time by merge tasks (the graph looks like the one in Figure 4.6 of Chapter 4, a reversed binary tree); finally, at the end of the iteration the main program post-processes the merged result, generating the current clusters that will be used in the next iteration. Consequently, if F is the total number of fragments, K-means generates F computation tasks and $F-1$ merge tasks per iteration.

In X10, the number of fragments for the points is also a parameter. Each fragment is created as a PlaceLocalHandle, assigned to a given place (the code is equivalent to that of the matrix multiplication in Figure 5.5(a)), lines 5-9) and initialised with random points. Every place possesses the same number of fragments. Regarding the main algorithm, in every iteration an activity is spawned to each place to perform the computation of its fragments. Inside each of these first-level activities, a second level of K activities, K being the number of fragments per place, are launched locally on each place to compute the clusters. This computation phase is enclosed by a finish construct. After that, there is a reduction phase, also inside a finish; here, every place sends K activities to place 0 to atomically accumulate the partial clusters. Similarly to what happens in the Java StarSs version, place 0 then performs a post-process whose result will be passed to the next iteration.

5.4.3 Programmability Discussion

Although ease of programming is something difficult to quantify, here we will take into account two possible indicators, namely the use of parallelisation syntax in the application and the number of lines of code.

On the one hand, Java StarSs applications are programmed in pure sequential Java, whereas X10 programmers need to use some constructs to express concurrency and data distribution. This analysis focuses on the X10 syntax that appears in our test applications, taking into account the following aspects:

- *Asynchronous computations*: X10 provides the async statement to launch new activities, where the user has to specify the place where the activity will run. In contrast, the selection of tasks in Java StarSs is done by means of a separate interface, leaving the sequential application untouched. The runtime automatically replaces on-the-fly the calls to the task methods by the creation of remote tasks and is in charge of scheduling them in the available workers.

- *Nested computations*: X10 allows to spawn activities from inside other activities and between any pair of places, thus making possible to express recursive algorithms and nested parallelism. This is used, for instance, in the K-means application to first send an activity from place 0 to every place, and then launch a second group of activities inside each place to compute the local clusters. The current Java StarSs implementation does not support such feature, and all the tasks are submitted from the main node to the workers.

- *Synchronisation*: synchronisation of activities in X10 has to be done manually, i.e. by enclosing them in a finish block. Contrarily, Java StarSs frees the user from task synchronisation: it is implicitly imposed by the data dependencies between tasks and enforced by the runtime; furthermore, even complex dependencies which would be hard to manage manually, like the ones in the Sparse LU kernel, are automatically detected by Java StarSs.

- *Data distribution*: X10 provides classes to create and initialise distributed structures, e.g. DistArray and PlaceLocalHandle. In a second step, activities can be submitted to the places where each part of these structures has been allocated in order to access them. On the other hand, the Java StarSs philosophy is not to let the programmer specify the distribution of the data, thus preferring simplicity of the programming model rather than the ability of tuning the performance of the application. Nevertheless, a Java StarSs task can be marked as an initialisation one, which means it will be scheduled in a round-robin fashion; such tasks can be used to allocate data directly in the workers, which overcomes the problem of being limited to the memory of the main node.

- *Data management*: in X10, mutable data (e.g. the parts of a distributed array) can only be accessed in its owner place. This means that, if an

Table 5.1: Number of code lines of the tested applications.

App. name	Java StarSs		X10
	Main Program	**Interface**	**Main Code**
Matmul	54	24	36
Sparse LU	128	57	203
K-means	148	41	135

activity needs some data that resides in a place other than its own, an explicit transfer must be issued (copyFrom method). An example of such scenario is the matrix multiplication benchmark when the three matrices are distributed: every update of a block of matrix C needs a block from A and a block from B, which may be located in different places. Moreover, X10 does not reuse data that has already been transferred (which also hinders performance). In contrast, the Java StarSs runtime is responsible for transparently copying the input data to the node where a task will run, possibly from another node where a predecessor task executed; in addition, the data are kept in the node for subsequent tasks to reuse them.

On the other hand, Table 5.1 shows the number of code lines for each application as another indicator of productivity. For Java StarSs, we distinguish between the number of lines of the Java sequential program itself ('Main Program') and those of the annotated interface that declares the tasks ('Interface'). Although the latter are not part of the main program, we include them for the sake of accuracy, since the parallelisation of the application is based on the information specified in the interface. Nevertheless, note that the length of the interface is not proportional to that of the main program - in fact, the interface is usually much shorter - and its definition is straightforward once the programmer has decided which methods will be remote tasks.

For matrix multiplication and K-means, the X10 programs are a bit shorter. The difference is mainly due to the initialisation of data structures. In X10, the initialisation is 'embedded' in the method which allocates the structure (e.g. in Figure 5.6(a), lines 4-5, the block elements are initialised to zeroes); this is more compact than the loop-based Java initialisation of the Java StarSs programs, but also less intuitive. However, if we compare the number of code lines of strictly the main loop e.g. in the matrix multiplication benchmark (Figure 5.4 for Java StarSs, Figure 5.5(b) for X10), that number is lower in Java StarSs.

In the case of the Sparse LU benchmark, the async, finish and array copies make the X10 version longer and remarkably harder to program.

In summary, based on the student's experience in programming in both languages, Java StarSs applications are arguably easier to code, while X10 requires the learning of some syntax which sometimes is not very expressive. On the one hand, Java StarSs frees the programmer from dealing with data distribution and transfer, spawning of asynchronous computations and synchronisation. On the other hand, the X10 constructs provide the programmer with more control over the application, making possible to fine-tune its performance.

5.5 Experiments

The experiments in this section will be divided in two series. The first series represents a continuation of the productivity comparison between Java StarSs and X10 started in Section 5.4, this time focusing on the performance of the applications introduced in that section. The second series extend the performance evaluation by running a standard parallel benchmark suite with Java StarSs, and comparing the results to other implementations of the same benchmarks.

5.5.1 Testbed

All the experiments were conducted in the MareNostrum supercomputer, hosted by the Barcelona Supercomputing Center. MareNostrum [9] is a cluster of 2560 JS21 blades, each of them equipped with two dual-core IBM PPC 970-MP processors that share 8 GBytes of main memory. Each core has a 64 KByte instruction/32 KByte data L1 cache and 1024 KBytes of L2 cache. The blades run the SLES10 (Linux) operating system. The interconnection network is Myrinet, accessible through the MX driver.

5.5.2 X10 Comparison Results

5.5.2.1 Test Setup

The IBM APGAS runtime was compiled to use the MX adaptor [112]. Since both Java StarSs and X10 share that same APGAS communication layer (see Figure 5.3), their performance can be directly compared in the tests.

For Java StarSs, the Java Virtual Machine used was the IBM J9 VM 1.6 for PPC 64 bits with its Testarrossa Just-in-Time (JIT) compiler. Regarding X10, an X10 program can be compiled to either Java or C++; in these tests we chose the X10 C++ runtime because it was able to work with more than one node. The X10 applications were compiled at the highest level of optimisation.

In the tests, for Java StarSs the application data (matrix blocks, points) were allocated on the worker nodes using initialisation tasks, so that those data were transparently distributed among the available nodes (see Section 5.2.5). In X10, the syntax for data distribution was used as described for each application in Section 5.4.2. The measures presented in the next subsection do not include this initialisation time neither for Java StarSs nor for X10.

The results are depicted for different numbers of cores. In the case of Java StarSs, the cores of the main node are not counted, given that in the current implementation that node never runs any task.

Concerning how the speedup is calculated, in Java StarSs the baseline is always a regular execution of the corresponding sequential Java application with the same granularity as when it is run with Java StarSs (e.g. same number of blocks and block size). In X10, the baseline is an X10 program which is equivalent to the corresponding parallel one (also with the same granularity), except that the activity-spawning and synchronisation constructs have been removed so that the execution is totally sequential.

Each application will be evaluated in three steps. First, an adequate granularity for the tasks/activities in Java StarSs/X10 will be selected. For a given input data size, small block sizes provide more parallelism but also more runtime overhead, because more tasks/activities are generated; conversely, using big blocks reduces runtime overhead but also limits parallelism and load balancing. Second, the chosen granularity will be used to analyse the scalability of the application in terms of execution times and speedup. Finally, other problem sizes will be explored.

5.5.2.2 Matrix Multiplication

The evaluation of the Matrix multiplication benchmark begins with a study of the best block size for the matrices. Figures 5.9(a) and 5.9(b) show, respectively, the execution times of the benchmark and the average running times of a task/activity (i.e. the multiplication of two blocks) for a range of block sizes. In these tests, the number of cores was fixed to 64 and the problem size was always the same, e.g. for a block size of 200x200 the matrix size (number of blocks) is 64x64, whereas for 400x400 the matrix is divided in 32x32 blocks. In X10, every place had a copy of the A and B matrices (first distribution described in Section 5.4.2.1).

At the light of the results, the best block size for both Java StarSs and X10 is 200x200 doubles. Such size is the last one in the considered range where the two input blocks of the block multiplication fit in the 1-MB L2 cache of each core, which explains the higher times for bigger block sizes. The increase in the execution time for smaller blocks can be attributed to the overhead of processing more asynchronous computations. As an example, provided that the Java StarSs matrix multiplication spawns a total of N^3 tasks, N being the number of rows or columns of the matrix, for $N = 64$ there is a total of 262144 tasks, while for $N = 128$ there are eight times more tasks. In the X10 version this overhead is a bit smaller because, as seen in Section 5.4.2.1, only N^2 remote activities are launched.

In a second series of tests, the chosen block size was used to analyse scalability. Figures 5.9(c) and 5.9(d) depict, respectively, the execution times and speedup of the benchmark for different numbers of cores. Furthermore, for X10, another kind of distribution was explored: in addition to ABRep (input matrices A and B are replicated), we present the results for ABDist (A and B distributed as detailed in Section 5.4.2.1).

Regarding the times, Java StarSs performs better than any of the X10 configurations. As can be seen in Figure 5.9(b), the Just-In-Time compiler of the Java Virtual Machine is able to produce faster code for the block multiplication than the X10 compiler and its C++ backend, which has a direct influence on the execution time of the benchmark. The speedup offers another view by comparing the times to the baseline execution. In this case, Java StarSs does not scale as much as X10 ABRep. The difference is due to three reasons. First, the execution for X10 ABRep is embarrassingly parallel and there is no data transfer at all between nodes, since every place has a copy of the whole A and

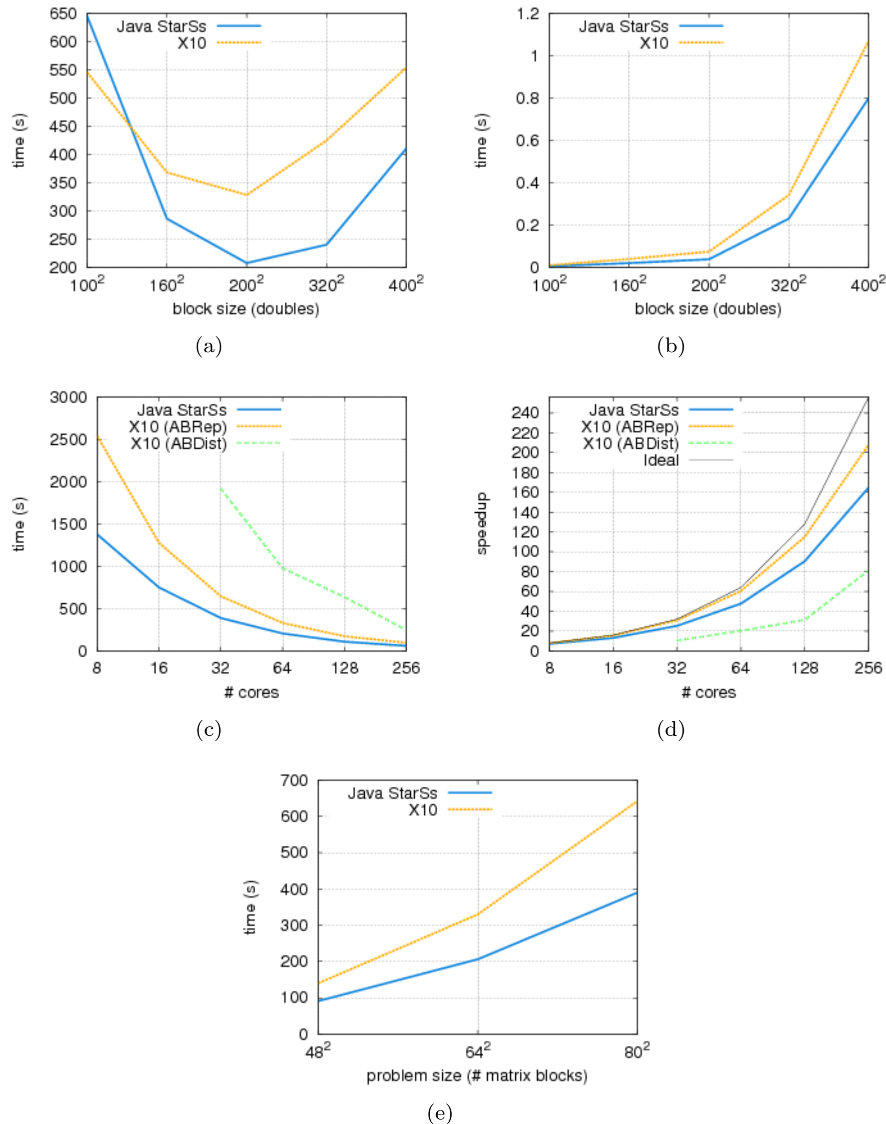


Figure 5.9: Test results for the Matrix multiplication benchmark for Java StarSs and X10. Study of the best block size, with a fixed number of 64 cores, keeping the same problem size and varying the block size: (a) benchmark execution times and (b) average task/activity times. Scalability analysis: (c) execution times and (d) speedup for a range of cores, input matrices of N=64 and M=200, i.e. 64x64 blocks of size 200x200 doubles; for X10, two different configurations of the matrices are considered: replicating matrices A and B (ABRep) or distributing them (ABDist). In (e), study of different problem sizes with a fixed number of 64 cores and using the best block size found (200x200).

B matrices and a part of the output matrix C; in Java StarSs, on the contrary, every worker allocates a portion of A, B and C, and consequently there is the need to eventually transfer A and B blocks, as well as the renamings created for C. Second, as mentioned earlier in this section, the X10 matrix multiplication only spawns N^2 remote tasks, while in Java StarSs this number is N^3. Third, the granularity of a block multiplication is smaller in Java StarSs than in X10 (Figure 5.9(b), for a block size of 200x200, 40 ms in Java StarSs versus 75 ms in X10); bigger granularities help scale better because the runtime has to process less tasks per unit of time and the workers need to be fed less often.

On the other hand, for the sake of performance X10 ABRep replicates the input matrices in every node, which implies more memory usage. As an alternative, the results for ABDist are shown, in order to see what happens when A and B are also distributed. In this case, the explicit transfers that must be added before every block multiplication significantly hinder performance; moreover, for 8 and 16 cores the results are not included because they failed to finish in a reasonable time, possibly due to the garbage collection of all the transferred blocks.

To conclude the study, other problem sizes were considered. In Figure 5.9(e), the execution times for three different sizes of the matrices are depicted, namely 48x48, 64x64 (the one already used in Figures 5.9(c) and 5.9(d)) and 80x80. The number of cores was 64 and the block size was the best found for both Java StarSs and X10 (200x200). The version taken for X10 is ABRep. The results show how, as the problem size increases, and so does the number of tasks/activities, the execution time in X10 grows faster than in Java StarSs.

5.5.2.3 Sparse LU

The Sparse LU benchmark represents a more challenging problem for the scheduling and dependency-analysis features of Java StarSs, due to the higher complexity of its task dependency graph. The numerous data dependencies and the different granularity of the various kinds of task make the load balancing harder.

The evaluation of Sparse LU proceeds in a similar way as with the Matrix multiplication. First, there is a study of the best block size for the matrix to be factorised. Figures 5.10(a) and 5.10(b) show, respectively, the execution times of the benchmark and the average running times of a task/activity (the `lu0`, `fwd`, `bdiv` and `bmod` methods) for a range of block sizes. In these tests, the number of cores was fixed to 64 and the problem size was always the same, e.g. for a block size of 300x300 the matrix size (number of blocks) is 64x64, whereas for 400x400 the matrix is divided in 48x48 blocks. In X10, the Block distribution described in Section 5.4.2.2 was used.

As can be drawn from the results, the best block size for X10 is 300x300 doubles; in Java StarSs the three smallest sizes present the same execution times, which made us select 300x300 as the block size for both. The effect of the cache on this benchmark is harder to analyse, since there are four kinds of task/activity that can work with one, two or three blocks; nevertheless, taking

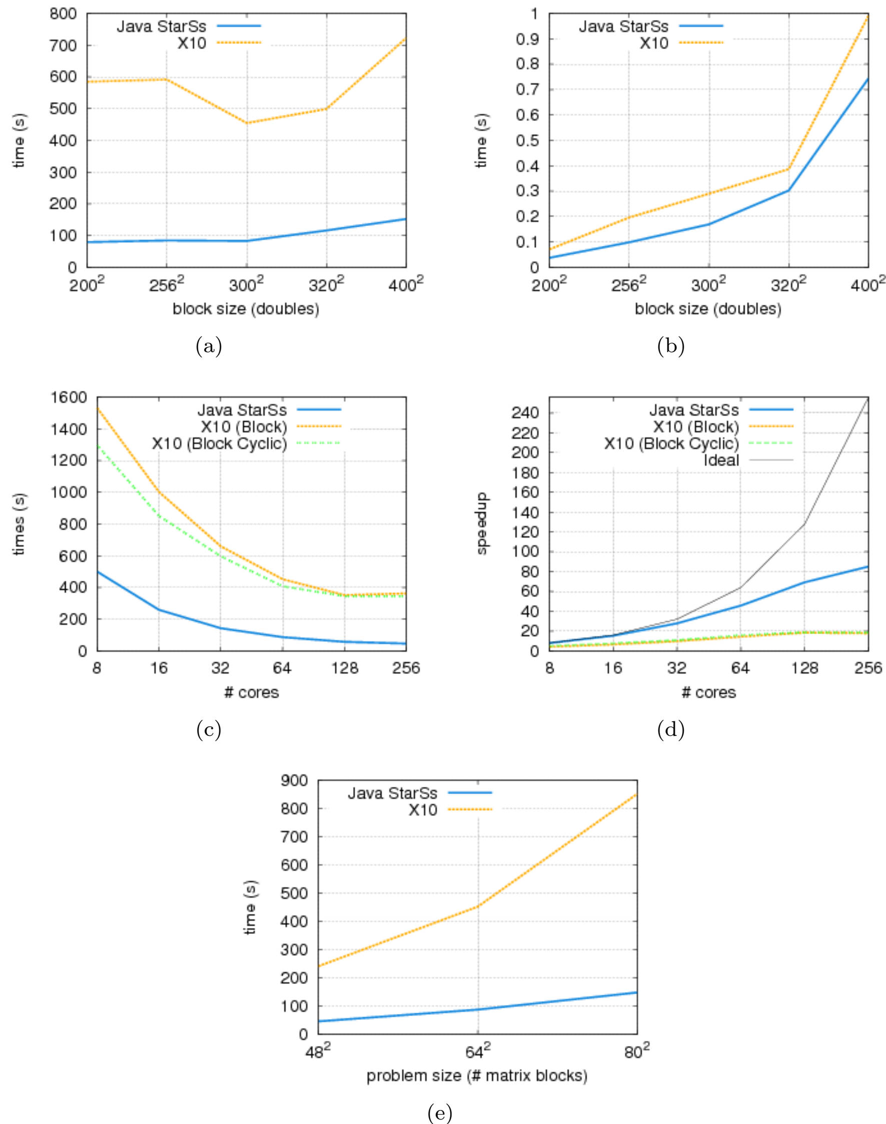


Figure 5.10: Test results for the Sparse LU benchmark for Java StarSs and X10. Study of the best block size, with a fixed number of 64 cores, keeping the same problem size and varying the block size: (a) benchmark execution times and (b) average task/activity times. Scalability analysis: (c) execution times and (d) speedup for a range of cores, input matrices of N=64 and M=300, i.e. 64x64 blocks of size 300x300 doubles; for X10, two different partitionings of the matrix to factorise are considered: Block distribution and Block Cyclic distribution. In (e), study of different problem sizes with a fixed number of 64 cores and using the best block size found (300x300).

into account Figure 5.10(b), 300x300 seems a good choice. The corresponding matrix size, 64x64, leads to the creation of 24510 remote tasks/activities.

In a second series of tests, the selected block size was used to analyse scalability. Figures 5.10(c) and 5.10(d) depict, respectively, the execution times and speedup of the benchmark for various numbers of cores. Furthermore, for X10, another kind of distribution is explored: in addition to Block, we present the results for Block Cyclic, also detailed in Section 5.4.2.2.

In this benchmark, Java StarSs has almost linear scalability up to 32 workers. Here, the data dependencies play an important role: the graph is complex and gets narrower from top to bottom. Therefore, at the end of the execution there can be worker starvation caused by the lack of tasks. This situation gets worse as we increase the number of workers.

Nevertheless, Java StarSs clearly outperforms X10 with respect to both execution time and speedup. There are at least four reasons that explain such results. First, the X10 version has some additional overhead caused by the explicit data transfers of input blocks, discussed in Section 5.4.2.2; in Java StarSs there are also transfers between workers, but every worker has an internal structure (the Application Data Directory, see Section 5.2.5) where it stores the data accessed by the tasks for eventual reuse. Second, a transfer in X10 is delayed until one of the worker threads of the node (which run the activities) gets free, whereas in Java StarSs there is a communication thread periodically polling for transfer requests on each worker. Third, Java StarSs is able to manage task dependencies in a more fine-grained fashion, whilst in X10 there are three coarse-grain synchronisation blocks per iteration. Fourth, as the application progresses, some blocks of the matrix are no longer accessed; since in X10 every datum can be modified only on its owner node, the owners of the unused blocks get starved. The Block Cyclic distribution mitigates a bit this effect, as can be seen in Figure 5.10(c), because the last rows of the matrix are cyclically assigned to places; however, at 128 cores (32 places) both distributions converge, each place receiving two consecutive rows of the 64-row matrix. Oppositely, in Java StarSs a written datum can be sent to any other node and updated again there, which helps balance the load.

Finally, other problem sizes were taken into account. In Figure 5.10(e), the execution times for three different sizes of the matrix are shown, namely 48x48, 64x64 (the one already used in Figures 5.10(c) and 5.10(d)) and 80x80. The number of cores was 64 and the block size was the best found for both Java StarSs and X10 (300x300). X10 ran with the Block distribution to ensure that the same number of matrix rows were assigned to each place. In these tests, the aforementioned drawbacks of the X10 Sparse LU make it again remarkably slower than the Java StarSs version as the problem size increases.

5.5.2.4 K-means

In order to parallelise the K-means application, the input points must be divided in fragments so that each computation task/activity calculates the clusters for a given fragment. In this sense, the evaluation of the fragment size is shown

in: 5.11(a) execution times of K-means for different sizes and 5.11(b) average running times of the tasks/activities that compute new clusters. In these tests, the number of cores was fixed to 64 and the problem size was always the same, e.g. for a fragment size of 500K points a total of 256 fragments were created, while splitting the points in 1024 fragments implies 125K of fragment size.

The study reveals that, differently from the other two applications, there is no fragment size that is equally suitable for both Java StarSs and X10. Regarding X10, the execution time of the application is quite stable across all the range of sizes, though 500K seems the best one. Conversely, in Java StarSs the fragment size makes the execution time vary significantly, the best value being 31.25K. The average task time, however, does not follow the same pattern and grows proportionally to the fragment size. Such irregular behaviour of the Java StarSs execution times depending on the fragment size will be explained next in the scalability analysis.

In order to do the evaluation of the application scalability, the two fragment sizes mentioned above were taken into account. Figures 5.11(c) and 5.11(d) depict, respectively, the execution times and speedup of K-means for various numbers of cores. The results of Java StarSs 31.25K show remarkable scalability and are very close to the X10 ones, which again are quite stable no matter the fragment size. Nevertheless, the 500K configuration is clearly not convenient for Java StarSs, mainly due to two factors: the influence of the Java JIT compiler and the bad load balancing at the end of each iteration.

Concerning the JIT effect, the method that finds new clusters for a given fragment of points is compiled at different optimisation levels during the execution of the application. Some of these levels apply profiling techniques and increase significantly the execution time of the method. At the highest level of optimisation the peak performance is reached, but it takes several executions of the method for that to happen. This means that, the more fragments are created, the more (and shorter) executions of the task method there will be, and thus the sooner the method will be optimised at the highest level. Figure 5.11(e) shows the execution time of the first ten iterations, using 64 cores, for both Java StarSs and X10 with the two fragment sizes. It can be observed how for Java StarSs 500K it takes longer to reach the peak performance, while for 31.25K this happens already in the second iteration. Note that X10 does not experience this problem, since the generated C++ code is statically compiled before execution and therefore the method duration is constant from the very beginning.

With respect to load balancing, it is more difficult to balance the load when dealing with coarse-grain tasks, e.g. the ones produced by Java StarSs 500K, which last about five seconds (Figure 5.11(b)). In a given node, one of the cores could be running such a long task while the others are idle because there are no more tasks to run. Since this balancing happens at the end of each iteration, where the results of all the computations are merged and the new clusters are passed to the next iteration, the influence in the overall execution time is more important. This explains why in Figure 5.11(e) the Java StarSs iteration time for the two fragment sizes becomes constant at different values.

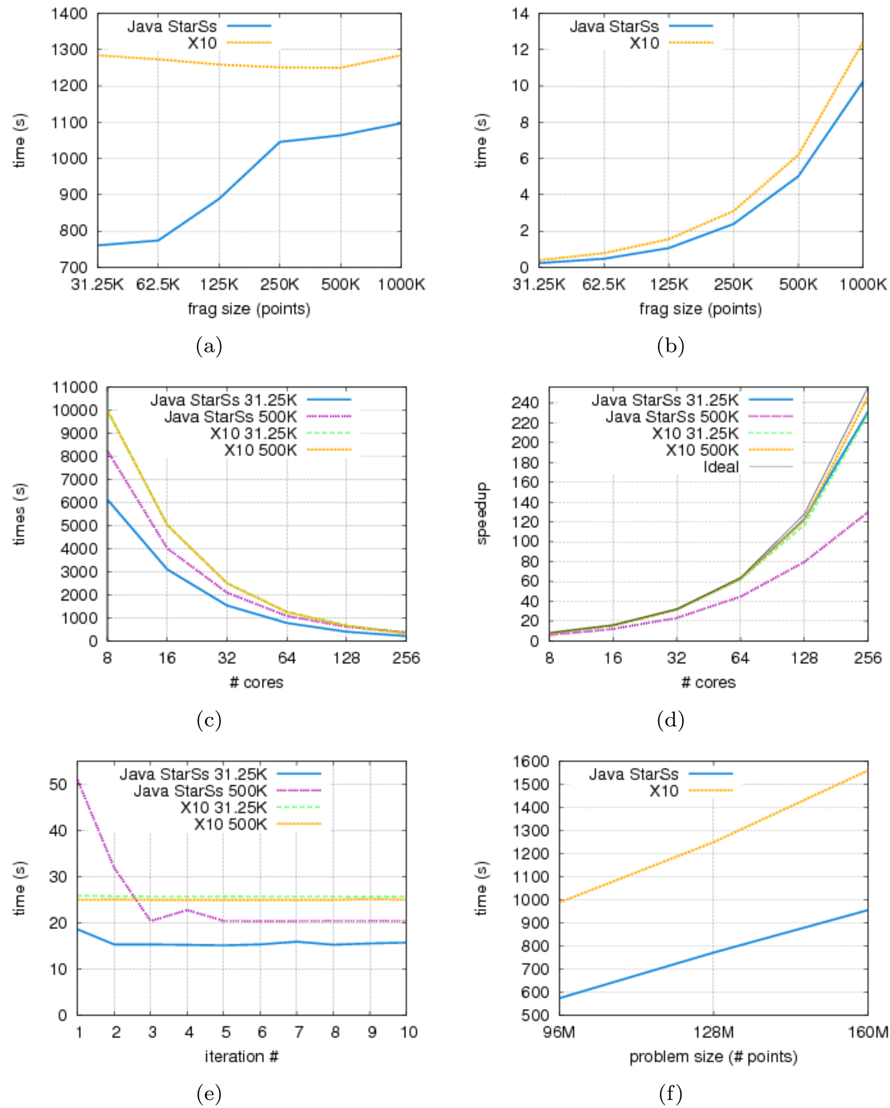


Figure 5.11: Test results for the K-means application for Java StarSs and X10. Study of the best fragment size, with a fixed number of 64 cores, keeping the same problem size and varying the fragment size: (a) application execution times and (b) average task/activity times. Scalability analysis: (c) execution times and (d) speedup for a range of cores, input parameters: 128000000 points, 4 dimensions, 512 clusters, 50 iterations; two fragment sizes are considered: 31250 points and 500000 points. (e) influence of JIT compilation in the iteration time for the two fragment sizes. In (f), study of different problem sizes with a fixed number of 64 cores and using the best fragment sizes found (31250 for Java StarSs, 500000 for X10).

Lastly, other problem sizes were studied. In Figure 5.11(f), the execution times for three different numbers of input points are depicted, namely 96000000, 128000000 (the one already used in Figures 5.11(c) and 5.11(d)) and 160000000. The number of cores was 64 and the fragment size was the best found for Java StarSs (31.25K) and for X10 (500K), respectively. The results show how the comparisons between Java StarSs and X10 still hold for other problem sizes, their execution times increasing more or less in the same proportion.

5.5.3 NAS Parallel Benchmarks

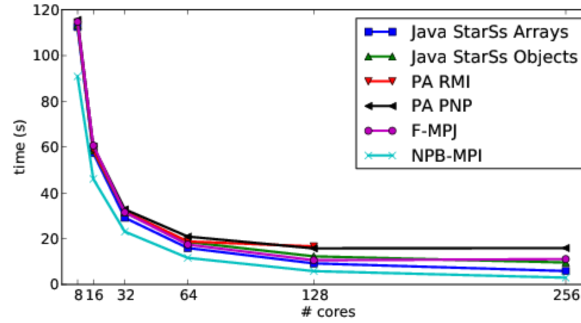
The NAS parallel benchmarks (NPB) [106] are a set of kernels that evaluate diverse computation and communication patterns, and they are widely used for parallel performance benchmarking. In order to extend the performance evaluation of the Cluster runtime, a sequential Java version of the NPB was developed for running them with Java StarSs. This section compares the results of the Java StarSs NPB with other implementations of the same benchmarks.

5.5.3.1 Test Setup

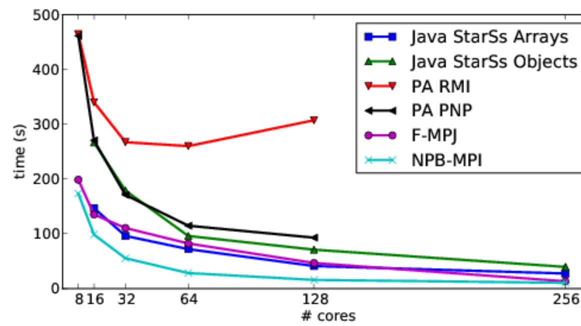
The Java StarSs implementation of the NPB was compared to three other versions of the benchmarks. The one used as reference was the MPI version (3.2, in C and Fortran) [106]. Besides, a couple of implementations in Java were considered as well. First, the version of ProActive [99], a parallel programming model and runtime for distributed-memory infrastructures that will be further discussed in Chapter 6; the ProActive NPB were executed over RMI [158] - its default communication protocol - and also PNP, a custom protocol created by the ProActive team. Second, a Java MPI version was also tested on top of F-MPJ [137], an MPI library for Java.

Most of the data exchanged in the benchmarks were implemented as 1-dimensional arrays, which can be transferred directly from the Java heap using the IBM APGAS runtime on Myrinet MX, with no extra copies; the lines named 'Java StarSs Arrays' in the plots represent the tests where such optimisation was used in our runtime. F-MPJ also exploits this optimisation on MX, which facilitates the comparison with 'Arrays'. On the other hand, we also ran the benchmarks instructing our runtime to treat the 1D arrays as regular objects; these tests correspond to the 'Java StarSs Objects' line of the plots. In this case, before transferring each object or array, it is first serialised into an array of bytes. Such marshalling happens in ProActive for both RMI and PNP, and therefore these tests were included for a more fair comparison. Regarding the NPB-MPI benchmarks, they were compiled with the xlc 10.1 and xlf 12.1 64 bit compilers with the -O3 option, and ran on a MPICH MX implementation.

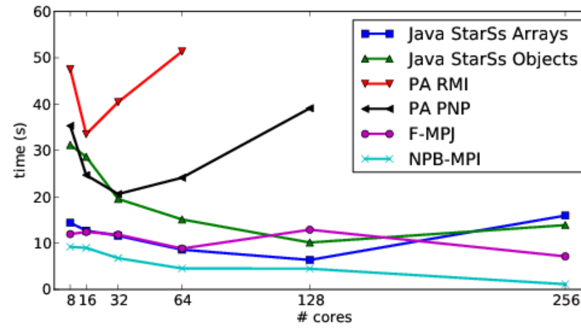
Concerning the common execution parameters, the kernels were run with C class size. The JVM used is the 1.6 IBM J9 64 bit. The times presented are the average of 5 executions for each number of cores; they do not include the time spent in initialisation and previous warm-up operations. All the cores in a node (4) were used.



(a) EP



(b) FT



(c) IS

Figure 5.12: Execution times (seconds) of the NAS parallel benchmarks: (a) Embarrassingly Parallel, (b) Fourier Transformation and (c) Integer Sort. Tested implementations: Java StarSs, ProActive, F-MPJ and NPB-MPI (original).

5.5.3.2 Embarrassingly Parallel (EP)

EP is a test for computation performance that generates pseudorandom floating point numbers and has few communications.

Figure 5.12(a) shows how the performance for all the Java versions is similar, and quite close to NPB-MPI. Nevertheless, ProActive is a bit behind, especially for RMI, which crashes when running on 256 cores. The next benchmarks will confirm the poor results of RMI, which can be partly attributed to the protocol itself and its scalability limitations.

5.5.3.3 Fourier Transformation (FT)

This benchmark tests computation performance by solving a differential equation with FFTs and communication by sending large messages.

As can be seen in Figure 5.12(b), 'Objects' outperforms PA PNP, which fails to run on 256 cores due to timeout errors. 'Arrays' also has a good behaviour and scales similarly to F-MPJ, but for 256 cores there is a sharp drop of the F-MPJ time, getting closer to NPB-MPI. This could be due to a change in the communication protocol: when the message size in F-MPJ is equal to or lower than 64 KB, it abandons the 'rendezvous' protocol to adopt an 'eager' one, which has no handshake; such transition happens with 256 processes, when the message size becomes 32 KB.

5.5.3.4 Integer Sort (IS)

This kernel also tests computation, but especially communication performance. It sorts a large array of integers and is characterised by numerous transfers.

In Figure 5.12(c), 'Objects' is definitely better than PA PNP, which does not seem to solve completely the scalability problems of RMI. Both 'Objects' and 'Arrays' experience a decrease in performance for 256 cores. The cause is the massive transfer of data in IS, combined with little computation. In the MPI versions, the data exchanges are implemented as all-to-all operations, where every process sends a distinct message to all other participating processes. Obviously, there is no such operation in sequential Java. Instead, in our implementation of IS, the exchange is done in two phases: first there are N^2 'get' tasks, N being the number of cores, where every core gets a piece of data from the rest; second, N^2 'set' tasks assign the values obtained by the gets. As N increases, so does the number of these types of task (e.g. 81920 for 256 cores), which produces more overhead for the runtime. This also happens in FT, but the fact that it is more computationally intensive implies coarser-grain tasks, which helps overlap computation and communication and consequently scale better.

The times for F-MPJ are quite the same as for 'Arrays' until 64 cores; from that point on, its behaviour is a bit irregular, increasing and decreasing again. The more performing implementation of the all-to-all exchanges in F-MPJ explains why it reaches better results than Java StarSs.

5.6 Related Work

The current mainstream parallel programming models in high-performance computing are OpenMP [102] and MPI [116]. Although OpenMP initially focused on loop-level parallelism for shared-memory systems, the last version 3.0 has been extended with a tasking model. Task data dependencies are not yet considered in this standard, although it is under consideration and there are proposals to extend it with them [107]. The MPI programming model has the widest practical acceptance for programming on distributed-memory architectures like clusters. MPI applications are composed of a set of processes with separate address spaces that perform computation on their local data and use communication primitives to share data when necessary. However, the common practice in MPI applications is to separate computation and communication in different phases, with the corresponding loss of performance due to load unbalance derived from the synchronisation points. An approach to overlap communication and computation is presented in [138] with a hybrid programming model that composes SMPSs with MPI. In such model, communications are encapsulated in tasks that can be aborted and re-scheduled when the communication is ready. This mechanism achieves a global asynchronous data-flow execution of both communication and computation tasks.

An alternative to these standards is Cilk [164], a task-based programming model. Cilk is based on the identification of tasks with the `spawn` keyword and the `sync` statement is used to wait for spawned tasks. Both OpenMP and Cilk consider nested tasks (tasks that generate new tasks) but data dependency detection is not supported and additional synchronisation points are required. While Cilk only supported parallel tasks, Cilk++ also supports parallel loops.

The Asynchronous PGAS languages X10 [101] (already introduced in Section 5.4.1) and Chapel [100] share the same underlying computational model as Java StarSs, which is based on spawning asynchronous computations as the main program executes. However, they expose that model to the programmer in fundamentally different ways. In both X10 and Chapel, the programmer is primarily responsible for decomposing the application's data across the partitioned global address space and for orchestrating the flow of computation through the system. Both languages provide high-level programming language constructs for describing data distribution and creating/synchronising large numbers of asynchronous tasks. In contrast, in Java StarSs managing data distribution and concurrency control is the responsibility of the underlying runtime system, not the programmer.

Swift [184] provides a scripting language to program parallel applications, as well as a runtime to execute them on large-scale clusters. The opportunities for parallel execution are revealed via a combination of parallel loop constructs and an implicit data-flow programming model. Calls to external programs from a Swift code are transformed into remote tasks and the dependencies between them are controlled. Although both Java StarSs and Swift are task-based dependency-aware programming models, they mainly differ in the kind of data they handle - all data types (files, arrays, objects, primitives) in

Java StarSs, only files in Swift - and in the language itself - pure Java in Java StarSs, scripting language with parallel statements in Swift.

StarPU [84] features a runtime system for executing applications on heterogeneous machines, i.e. equipped with accelerators such as GPUs. In StarPU, the programmer can define a 'codelet' - an abstraction of a task - that can be executed asynchronously on a core of the machine or offloaded to an accelerator. Similarly to Java StarSs, the programmer specifies the direction of the codelets parameters so that the runtime discovers and enforces the dependencies between them. Differently from Java StarSs, which does not require to change the sequential code of the application, StarPU programmers need to include some API calls in the code for task spawning or data registering. The StarPU runtime cannot work with more than one node like Java StarSs, but it is able to move data between different computational units and take into account data locality when scheduling tasks.

Terracotta [59] is a solution for running Java web applications on clusters. Like Java StarSs, Terracotta relies on dynamic bytecode instrumentation to control the execution of the application. The programmer creates and synchronises Java threads as normal, but those operations are transparently transformed by Terracotta into their distributed version when necessary; however, Terracotta has no concept of global thread scheduler and the programmer must manually launch multiple instances of the application and balance the load. Regarding the application data, the programmer specifies in a configuration file those classes to be shared between nodes, which will cause Terracotta to populate the changes made to instances of these classes. Unlike Terracotta, Java StarSs is based on fully-sequential Java programming and a shared-nothing paradigm, which frees the user from managing threads and data sharing; furthermore, the load of the application (the tasks) is automatically balanced.

5.7 Summary

This chapter has provided an overview of the second infrastructure contemplated in this dissertation: Clusters. Such infrastructure is mainly characterised by resources with fairly homogeneous hardware and software, interconnected by dedicated and fast local-area networks. Applications that run on clusters can benefit from those high-speed links to perform communications of data structures involved in the computation.

In order to fully exploit the characteristics of Cluster computing, the Java StarSs runtime was re-designed and implemented to move away from Grid technologies and adapt to the new scenario. In that sense, the IBM APGAS runtime played a key role, becoming the underlying communication layer of Java StarSs and enabling fast one-sided communications - with active messages - between nodes. Moreover, the new design featured persistent workers that maintain a cache of task data, thus favouring data reuse and locality, and that are able to exchange data without the intervention of the main node, which increases scalability.

Regarding the programming model, it was extended to support memory data structures (arrays, objects) as task parameters. However, in order to preserve the simplicity of the model, it was decided not to allow the programmer to explicitly specify the distribution of those data, as it happens in the APGAS languages. Instead, the model permits to mark tasks as initialisation tasks, which are scheduled in round-robin among the available resources and thus can be used to uniformly allocate data on those resources; furthermore, thanks to those data-allocating tasks that create data directly in the workers, the total amount of memory is extended to that of all the nodes, which favours scalability.

In the evaluation part of the chapter, the Cluster flavour of Java StarSs has been analysed in terms of productivity, i.e. with respect to its programmability and performance, showing a good tradeoff between these two aspects. On the one hand, the ease of programming has been compared to that of the X10 language, concluding that the use of X10 entails learning a compact but sometimes opaque syntax, while Java StarSs applications only require knowledge of sequential Java and they are arguably easier to code. Java StarSs frees the programmer from dealing with data distribution and transfer, spawning of asynchronous computations and synchronisation; in contrast, X10 constructs provide the programmer with more control over the application, making possible to fine-tune its execution.

On the other hand, this chapter has also focused on performance, comparing the results of Java StarSs and X10 for a set of applications. As a general conclusion, Java StarSs performs better than X10 in applications where some data is both read and written, and often a given piece of data updated on one node needs to be read by another node. Such applications usually have complex data dependencies which are hard to control manually in the code. Java StarSs handles those cases with no burden for the programmer, creating a task dependency graph, defining renamings of written data, transferring them if necessary to balance the load and caching them for later use. On the contrary, when the user can partition the application data in such a way that every node only accesses its own fragment or few transfers are required, X10 can manage it more efficiently; furthermore, the possibility of spawning asynchronous computations between any pair of nodes in X10 contributes to reduce the load of place 0 (main node). As a complement, the study of the well-known NAS parallel benchmarks demonstrates that Java StarSs provides competitive results compared to other Java-based implementations and the reference Fortran/C code. However, since it is based on sequential Java to simplify programming, Java StarSs lacks collective communication operations that exist e.g. in MPI; hence, communication-intensive applications with little computation constitute a case where Java StarSs still has room for improvement. The possible extensions and modifications to the Java StarSs design in order to increase its performance and scalability even further will be discussed in the Conclusions Chapter 7.

Chapter 6

Cloud

This chapter completes the trilogy of infrastructure chapters with the currently emerging trend of *Cloud computing*. Clouds, just like grids and clusters, present a set of distinctive characteristics that motivated changes and extensions in the Java StarSs programming model and runtime. In that sense, Java StarSs evolved once more to exploit the service-oriented nature and virtualisation features of the Cloud.

The content of the chapter will be organised in the following points: first, an introduction to the context of Cloud computing and to some basic concepts; second, an explanation of the runtime design decisions driven by the scenario; third, a description of the technologies that influenced the runtime implementation for clouds; fourth, a programmability evaluation of the programming model, focusing on the use of services and objects in the model and comparing Java StarSs to other approaches; fifth, the results of the experiments carried out in private and public clouds; finally, a related work section and a concluding summary.

6.1 Context

6.1.1 Cloud Computing

According to the National Institute of Standards and Technology (NIST) [141], Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g. networks, servers, storage, applications and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

Clouds are now emerging as an IT (Information Technology) paradigm shift, challenging the common understanding of the location, management and economics of IT infrastructures. The next concepts are pillars of Cloud computing:

- *Everything as a service*: clouds enable technology to be accessed as services delivered over the Internet.

- *Utility computing*: pools of computing resources are consumed and paid by users as they need them.

- *Virtualisation*: virtualisation technologies introduce a layer between the execution environment - seen by applications and operating systems - and the hardware underneath [115]. Virtual Machines (VMs) are representations of physical machines with their own set of virtual hardware and software; multiple VMs of different users can be multiplexed and isolated from each other in a single physical machine, which makes resource management more efficient.

- *Elasticity*: Cloud resources can be elastically acquired and released, in some cases automatically, in order to scale with demand.

Companies can outsource to the Cloud any part of the IT stack. The levels of that stack, which ranges from hardware to applications, are known in the Cloud as [141] (see Figure 6.1):

- *Infrastructure as a Service (IaaS)*: the consumer is provided with virtualised basic computing resources like processing, storage and network. The consumer can select the amount and configuration of those resources, where she can deploy and run software of her choice (operating systems and applications) while the underlying Cloud infrastructure is kept transparent. A pioneer and leading player in this sector is Amazon, which started offering the Amazon Elastic Compute Cloud (EC2) [2] in 2006, permitting customers to rent VMs in a pay-as-you-go basis; other examples are Rackspace Cloud [52] and FlexiScale [17].

- *Platform as a Service (PaaS)*: the consumer is provided with programming models, libraries, services and tools to ease the development, testing and deployment of applications onto a Cloud infrastructure. Such infrastructure is not managed by the consumer, who only controls the configuration of the deployed applications and their hosting environment. Examples in this field are Google App Engine [20], Microsoft Azure [37] and Salesforce.com's Force.com [18]. The Java StarSs programming model and runtime also fall into this category, since they offer means to program and run applications in the Cloud.

- *Software as a Service (SaaS)*: the consumer is provided with network-accessible applications running on a Cloud infrastructure. The applications can be invoked from diverse client devices through, for instance, a web browser or a program interface. The consumer has no control over the Cloud infrastructure nor the application capabilities, although she may be allowed to define user-specific application settings. Examples in this area are Salesforce.com sales management applications [53], Google's Gmail [22] and NetSuite [38]. Applications programmed with the Java StarSs model and published as a service can be considered SaaS as well.

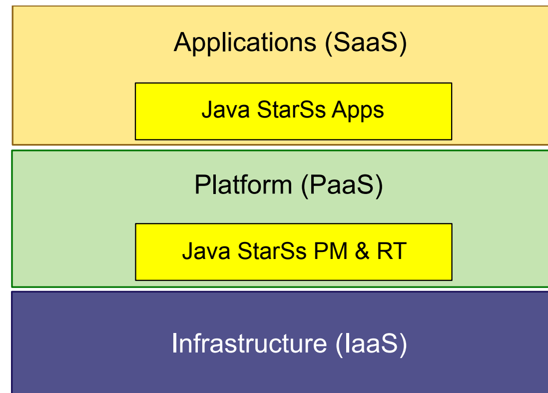


Figure 6.1: Location of the Java StarSs programming model, runtime and applications in the Cloud stack.

An organisation that leverages Cloud computing does not need, therefore, to own the IT infrastructure, platform or services; instead, these can be hosted by a third-party and delivered over the Internet, which allows to reduce infrastructure costs and maintenance effort. This was the initial idea behind Cloud computing - what is known as *Public Cloud* - but some companies are reluctant to adopt it because they do not want to lose control over their data or they prefer to own their resources. As a response, *Private Clouds* install Cloud technologies on-premises and provide services to internal users; this way, the company keeps its data at home but still benefits from e.g. flexible resource management, better hardware utilisation and metering [70]. A hybrid approach can also be taken: a private cloud can be complemented with public cloud resources to face peaks in load - what is called *Cloud Bursting*.

In conclusion, enterprises working with the Cloud can adjust virtualised resources in real time to meet demand; this is especially important when such demand fluctuates significantly, and it is potentially more cost-effective than over-provisioning local resources. Moreover, by facilitating and accelerating software development, the time to market of new products decreases.

6.1.2 Clouds and Service-Oriented Architectures

The growing interest in providing applications as Cloud services raises one question: how to develop such applications in order to take full advantage of the service-oriented nature of the Cloud? A potential answer is the *Service-Oriented Architecture* (SOA) paradigm [135].

SOA defines the architecture of an enterprise system as a set of services delivered to a network of consumers. Those services are loosely-coupled and correspond to specific business processes; furthermore, they can be combined to create new *composite services with an added value*. Hence, SOA favours software reuse and increases agility when adapting to changes. A company willing to

embrace SOA needs to decompose the architecture of its system into functional primitives, understand their behaviour and the information they receive/generate and finally re-build the system by defining language-neutral service interfaces [157]. In order to support this architectural style, the Web Service standards [68] provide a fairly mature and predominant implementation of the SOA concept.

In the last years, the use of SOA has been mainly restricted to internal integration in companies rather than exposed for external consumption [150].In that regard, Cloud computing represents a new field where SOA principles could be applied on an Internet scale, given the service-orientation of both paradigms. An organisation may decide to rely on Cloud-resident services e.g. to outsource a part of its SOA business logic.

The service-oriented nature of Cloud computing - especially when complemented with SOA - brings a need for, on the one hand, programming models that ease the development of applications composed by services and, on the other, systems that orchestrate (i.e. manage, steer) the execution of those services in the Cloud. In that sense, the Java StarSs programming model and runtime were extended to support service invocation, composition and orchestration in Cloud environments, which will be further discussed in Section 6.2.

6.1.3 Clouds for HPC Science

Scientific applications are characterised by an ever-growing need for processing, storage and network resources. So far, such need has been addressed by either Grid or Cluster computing, discussed in Chapters 4 and 5, respectively.

Research projects or institutions with enough money are able to purchase their own cluster, dedicated to satisfy the computing demands of their users and applications. In this case, there is a fixed computing capacity that can only be extended or upgraded by buying more resources.

When local resources do not suffice and new equipment cannot be afforded, one can opt for the resource-sharing philosophy of grids: an organisation that contributes to a grid with its local resources gains access to the overall infrastructure, while still keeping the ownership of its resources and being able to decide how to share them with others, like in the Open Science Grid (OSG) [44]. In other cases, research projects can simply apply for compute cycles on national or international Grid initiatives such as the European Grid Infrastructure (EGI) [14]. However, quality of service is generally not guaranteed in grids, and consequently an application may have to wait for resources when it needs them [105].

Currently, Cloud computing is being investigated as an alternative to owner-centric HPC, although the field is still in its infancy. The pros to adopt the Cloud for e-Science include elasticity to respond to peaks in resource demand, immediate provision of resources and cost-effectiveness of the pay-per-use model. On the cons side, the low performance of VMs in comparison with physical nodes, mainly due to virtualisation and sharing of the underlying infrastructure in the Cloud provider [125].

Another aspect in favour of Cloud computing for scientific applications is its flexibility when it comes to installing and configuring different environments and technologies in the VMs: users can deploy VMs with arbitrary software (operating system, applications, libraries). In contrast, grids require skilled system administrators to manage the process of maintaining and upgrading the infrastructure for particular communities of users. Besides, there are often constraints regarding e.g. the operating system or the middleware to use (Scientific Linux and gLite in EGI, respectively). Ideally, user communities should be able to deploy technologies or updates to their software environments at a timescale that suits them, and this is something that the Cloud can make possible. As a matter of fact, some Grid initiatives are already starting to study how they could attract new users with Cloud computing [75].

In summary, clouds have some advantages that could be exploited by computational science, but this field is still largely unexplored. In this chapter, we will contribute with an example of a service-based e-Science application programmed with Java StarSs and executed in the Cloud (Section 6.5).

6.2 Runtime Design

Chapter 5 explained how the design of the runtime evolved from Grid to Cluster in order to address the differences between these two scenarios. Likewise, the characteristics of Cloud computing, enumerated in Section 6.1.1, also motivated significant changes in the design of the Java StarSs Grid runtime.

The next subsections overview the new aspects of the Cloud runtime. Figure 6.2 shows the whole picture of this new design and can help follow the explanations.

6.2.1 Support for Services as Tasks

In Java StarSs, only sequential programming skills are required to write applications composed of services. In Chapter 2, Section 2.3.2 showed how service operations can be easily invoked as normal methods from a Java StarSs application, with no use of any library or new syntax. Those operations can be selected as tasks and thus be integrated in the data flow of the application, consuming/producing data from/to other tasks or the main program.

Such feature at programming model level required some new support from the runtime:

- Allow for services to be integrated in the data dependency control, synchronisation and scheduling mechanisms. The runtime can now add service tasks to the dependency graph, possibly together with method tasks, and orchestrate the execution of such tasks in the available resources. Besides, it also watches the accesses from the main program to data produced by a service operation.

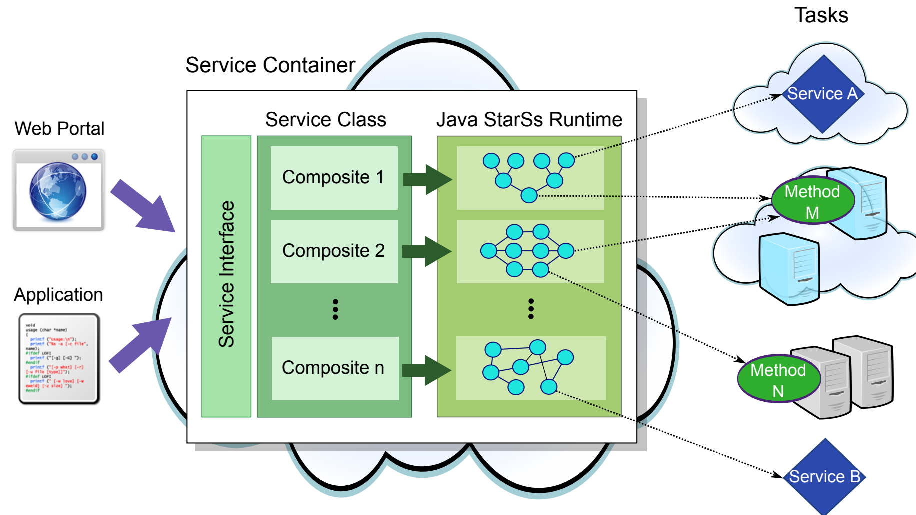


Figure 6.2: Architecture of the Java StarSs Cloud runtime. A service hosted in a Web services container can be accessed by any service consumer (e.g. web portal, application). The interface of this service offers several operations, which can be composites previously written by a service developer following the Java StarSs programming model. When the container receives a request for a given composite, the Java StarSs runtime starts generating the corresponding task dependency graph on the fly, so that it can orchestrate the execution of the selected tasks. Service tasks will lead to the invocation of external services (possibly deployed in the Cloud), while method tasks can be run either on virtualised Cloud resources or on physical ones.

- Implement the invocation of service operations inside the runtime, for it to act on behalf of the application and call a service operation when freed from dependencies.
- Extend data management to support object orientation, since objects are often parameters of service operations. This means enabling tasks to handle objects, detecting dependencies on them and synchronising their access from the main program.

6.2.2 Integration In a Service-Oriented Platform

Composing and orchestrating an application that invokes external services is only the first step to service orientation. For such an application to become a SaaS composite, it must be also deployed and published as a service with an added value.

In that sense, the second step consisted in fully integrating the runtime in a service platform. The next points summarise the main architectural changes:

- Publication of the composite: a class containing the implementation of one or more composites (like the one in Figure 2.5(b), Chapter 2, Section 2.3.1) first goes through the instrumentation phase, which makes the composites invoke the Java StarSs runtime for task creation and data synchronisation (see Chapter 3, Section 3.2). After that, the class is included in a service package and deployed in a service container, which can be hosted in the Cloud. Hence, composites become service operations *published in a service interface* so they can be accessed by service consumers.

- Concurrent applications: a fundamental change with respect to the Grid and Cluster scenarios is the ability of the Java StarSs runtime to *manage more than one application concurrently.* When the service is deployed, the Java StarSs runtime is started and awaits the arrival of new work. Multiple requests for the execution of one or more composites can then reach the service container, whose threads begin to process them in parallel. This makes the Java StarSs runtime receive task creation requests coming from several composite executions; as a response, the runtime builds a task dependency subgraph for each of them. Furthermore, the subgraphs of different composite executions can be connected if they access shared data (e.g. some structure declared in the service class or a file).

- Service nesting: the new service-oriented design of Java StarSs inherently leads to *nested services.* As explained earlier in this section, the orchestration of a Java StarSs composite can include invocations to external services. Those services, at their turn, can also be composites deployed in another service container and managed by another Java StarSs runtime. This creates a SOA where, on the one hand, services rely on other services to provide a new functionality and, on the other, a hierarchy of orchestrations is unrolled as the execution progresses.

6.2.3 Exploitation of Virtual Cloud Resources

In the final step of its Cloud adaptation, the resource management of the Java StarSs runtime was extended to handle virtual machines hosted in the Cloud.

In addition to external services, the nodes of a composite's task dependency graph can also correspond to Java methods. Chapters 4 and 5 showed how the runtime was able to schedule those method tasks in a given set of Grid and Cluster resources, respectively. In order to exploit the virtualisation features of Cloud computing, the new design incorporates the ability to reserve VMs and submit method tasks to those VMs. Moreover, Cloud elasticity is exploited by increasing/decreasing the number of VMs depending on the current load.

The Java StarSs runtime communicates with the Cloud by means of Cloud connectors. Each connector implements the interaction of the runtime with a given Cloud provider, more precisely by supporting four basic operations: ask for the price of a certain VM in the provider, get the time needed to create a VM, create a new VM and terminate a VM. Connectors abstract the runtime

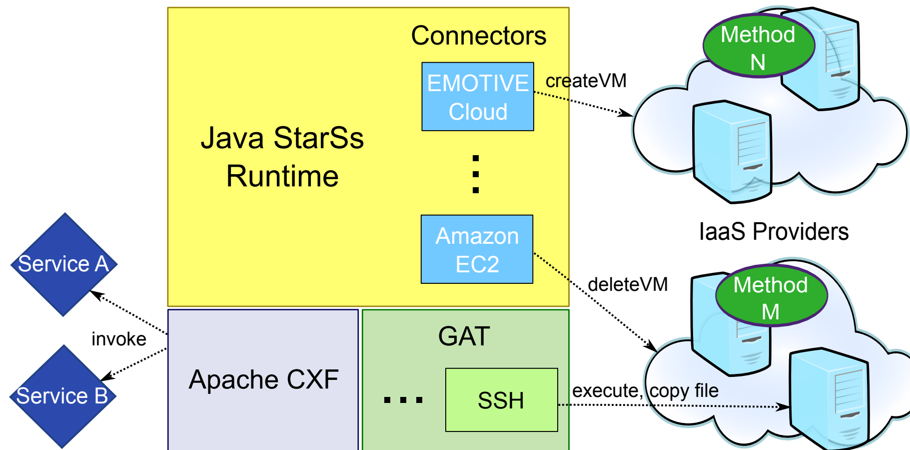


Figure 6.3: Technologies leveraged by the Java StarSs Cloud runtime.

from the particular API of each provider; furthermore, this design facilitates the addition of new connectors for other providers.

The task load generated by the execution of composites directly influences the number of VMs acquired. The runtime calculates the current load by inspecting the number of dependency-free tasks and their estimated time of execution; the time estimation for a task takes into account previous executions of the same kind of task. The runtime may decide to increase the number of VMs based on the task load, the current VMs and the time that it takes to launch a new VM. Similarly, when the number of tasks to execute decreases, the runtime may terminate a VM. The provider where to create/eliminate a VM is chosen depending on the cost of a VM in all the available providers. Finally, the runtime also takes into account task constraints when requesting the capabilities of a new VM.

On the other hand, this new resource management makes possible to have hybrid executions that combine physical machines (like for grids and clusters) and virtual ones that can scale on demand, depending on the load produced by method tasks. Please note that service tasks do not take part in the elasticity mechanism, since they are executed in external service containers whose resources are not under the control of the Java StarSs runtime.

6.3 Relevant Technologies

This section presents the Cloud and SOA technologies that were used to implement the runtime design seen in Section 6.2. Figure 6.3 illustrates these technologies.

6.3.1 Cloud Provider Connectors

The dialog of the Java StarSs runtime with the different Cloud infrastructure providers is encapsulated inside connectors, each containing the particular API calls to manage resources in a certain provider.

The connectors that will be tested in this thesis are the ones for EMOTIVE Cloud [176] and Amazon EC2 [2], although there exist prototypes for other offerings like OpenNebula [46] and Microsoft Azure [37].

6.3.2 SSH Adaptor of JavaGAT

Once a VM has been acquired from a provider, a key set is configured in order to access that VM through SSH. As explained in Chapter 4, Section 4.3, the Grid runtime of Java StarSs was built on top of the JavaGAT API [78], which features adaptors for different kinds of Grid middleware.

Since one of those adaptors implements the SSH protocol, the Cloud runtime reuses that technology and contacts VMs for job submission and file transfer by SSH with JavaGAT. Moreover, it is still possible to access physical resources with SSH or other adaptors as well.

6.3.3 Apache CXF

Regarding service tasks, Java StarSs executes them by means of Apache CXF [5], an open source services framework that helps build and develop services using programming APIs. One of the APIs implemented by CXF is the Java API for XML Web Services (JAX-WS), which can be used to program clients of SOAP-based web services.

Hence, the Java StarSs runtime utilises CXF to create dynamic clients that, given the WSDL of the server, namespace, port name, operation name and parameters, generate a SOAP message to invoke an operation of a service. This way, the runtime can request the execution of service tasks to external servers on behalf of the application.

6.4 Programmability Evaluation

This section will evaluate the ease of programming of Java StarSs in Cloud and service-oriented environments. For that purpose, a comparison with other approaches in the same field will be carried out, in particular by implementing the same applications in the models/languages examined and then highlighting the most relevant differences.

In a first subsection, the development of composite services will be addressed, comparing Java StarSs to the WS-BPEL language [40]. After that, the use of objects in parallel programming models will be illustrated by contrasting Java StarSs with ProActive [99]; this second study is included in this chapter because services often manipulate data in the form of objects, although the use of objects with Java StarSs is absolutely possible in Grid and Cluster environments as well.

6.4.1 Programming with Services

6.4.1.1 WS-BPEL

WS-BPEL [40] is a workflow-based composition language for web services, standardised by the OASIS consortium [47]. A WS-BPEL composition is an XML document (Extensible Markup Language [16]) whose tags represent different actions:

- *Variable definition*: the 'variable' tag defines a variable of a type specified in an XML schema.
- *Structure*: 'sequence' encloses an ordered sequence of steps, whereas the statements encompassed by 'flow' can be executed in parallel. In order to represent data dependencies between statements inside 'flow', WS-BPEL provides the tag 'link'.
- *Service interaction*: the 'receive' and 'reply' tags are placed at the beginning and end of the composition to define the input and output of the composite service, respectively. The 'invoke' tag represents a call to a external service operation defined in a WSDL file.
- *Control flow*: WS-BPEL provides tags like 'if'/'else' or 'while' to express control-flow statements as in an imperative language.
- *Embedded code*: 'javaCode' allows to embed Java statements in WS-BPEL in order to, for instance, check if a condition holds.
- *Others*: other examples include the 'assign' tag to copy data from one variable to another, or 'throw' to raise exceptions.

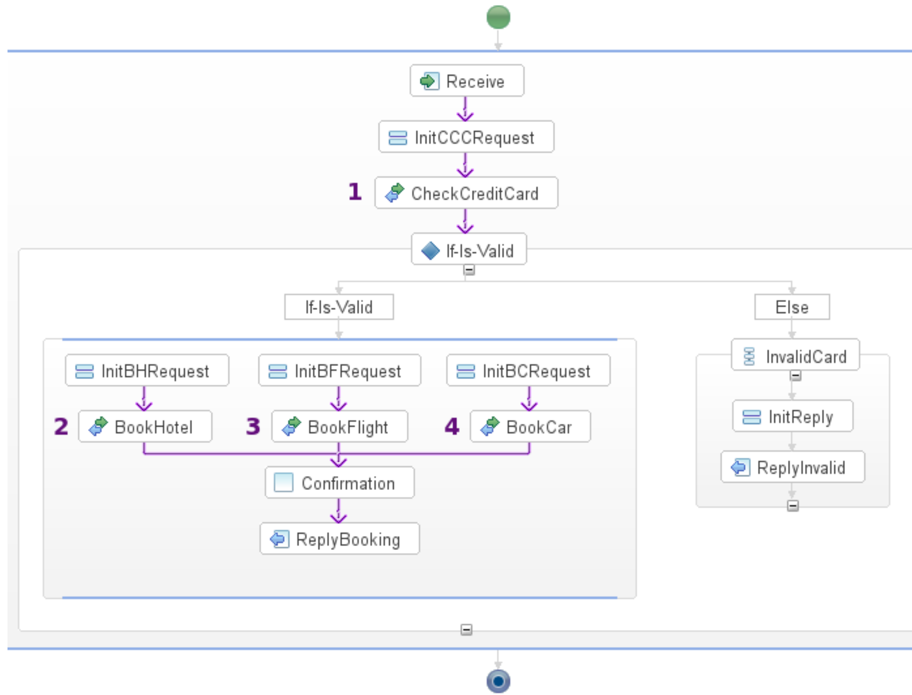
Since editing XML documents is a tedious task, some visual editors for WS-BPEL have appeared; one of them is the Eclipse BPEL Designer [12], which will be used in this section.

6.4.1.2 Travel Booking Service

The comparison between Java StarSs and WS-BPEL will be illustrated by an example derived from the one in [65].

The scenario of this example is based upon the procedure of making travel arrangements. Therefore, the composite service to be built will go through the steps of a travel booking process, which include:

1. The customer enters the data for her travel arrangements.
2. The system checks her credit card information.
3. If the card validation succeeds, three different reservations are made for the flight, hotel and car; otherwise an error is returned.
4. Once the reservations have finished, a confirmation number is returned to the customer.



(a)

```
1  <bp:process name="TravelBooking" ...>
     ...
2    <bp:flow ...>
       ...
3      <bp:invoke name="BookFlight" operation="bookFlight"
4                 portType="ns4:FlightReservation"
5                 inputVariable="FlightReservationInput"
6                 outputVariable="FlightReservationOutput" ...>
7        <bp:targets> <bp:target linkName="Link6"/> </bp:targets>
8        <bp:sources> <bp:source linkName="Link9"/> </bp:sources>
9      </bp:invoke>
       ...
10   </bp:flow>
11 </bp:process>
```

(b)

Figure 6.4: In (a), graphical workflow of the travel booking composite, as shown by the Eclipse BPEL Designer; the invocations to external services are numbered. In (b), a fragment of the corresponding WS-BPEL document, focusing on the invocation of service BookFlight.

6.4.1.3 Comparison

The graphical representation of the travel booking composite service, captured from the Eclipse BPEL Designer, is depicted in Figure 6.4(a). The arrows represent data dependencies (links) between WS-BPEL statements. Inside the composite's body, four external services are invoked: (1) CheckCreditCard, (2) BookHotel, (3) BookFlight and (4) BookCar. Before the invocation of each of those services, an 'assign' statement initialises a request variable (input of the service). CheckCreditCard is executed first to ensure that the card is valid. After that, there is an 'if' clause that checks the result of the validation: if it succeeded, the three reservation lines are started in parallel, otherwise the composite replies with an error. The three reservations converge in the Confirmation box, which represents a Java code embedded in the WS-BPEL that checks the proper completion of the reservations and sets a variable with the booking confirmation number to be returned to the customer.

In Figure 6.4(b), a part of the corresponding WS-BPEL is shown, in particular the one that contains the invocation to the BookFlight service - the complete document is about 325 lines long. In lines 3-6, the 'invoke' element specifies some information about the operation called (e.g. name of the operation and port, input and output variables). Also, inside 'invoke' there are the 'targets' and 'sources' tags (lines 7 and 8) that define the links of that service invocation with other activities in the flow: BookFlight depends on the InitBFRequest assignment and produces data consumed by Confirmation ('Link6' and 'Link9', respectively). In contrast, Java StarSs does not require to manually specify the data flow; instead, data dependencies between tasks and synchronisation from the main program are performed transparently to the programmer.

The Java StarSs version can be found in Figure 6.5. In (a) appears the composite service operation bookTravel, which is a method of a service class; the whole class is about 80 lines long. The code of bookTravel follows the same steps as the WS-BPEL implementation, which were described in Section 6.4.1.2.

In the Java StarSs composite, the travel booking process is implemented as a sequential Java program, and the invocations to the external services are regular Java method calls (underlined in Figure 6.5(a) and selected in (b)); before those calls, initialisation methods build the request objects to be passed to the services. Like in WS-BPEL, the calls to the reservation services are executed in parallel: the Java StarSs runtime will spawn an asynchronous and independent task for each of them. Regarding control flow statements, like the 'if' that checks the result of the card validation, they are Java statements in Java StarSs, as opposed to the XML tags in WS-BPEL.

Finally, in the confirmation phase (lines 13-18 in Figure 6.5(a)), the Java StarSs runtime automatically synchronises the main program with the results of the reservation services, so that the confirmation number can be generated if everything went well. For WS-BPEL, the Confirmation box in Figure 6.4(a) corresponds to an embedded Java code that is equivalent to that in Figure 6.5(a), lines 13-18. Therefore, the users of WS-BPEL still need to use an imperative language to cover some cases that WS-BPEL alone cannot handle.

```
1   @Orchestration
2   public BookResponse bookTravel(BookRequest tbRequest) {
        // Check credit card
3       CCardRequest ccRequest = initCCCRequest(tbRequest);
4       CCardResponse card = checkCreditCard(ccRequest);

5       BookResponse replyBooking = new BookResponse();
6       if (card.isValid()) {
            // Hotel booking
7           HotelRequest hrRequest = initBHRequest(tbRequest);
8           HotelResponse hotel = bookHotel(hrRequest);

            // Flight booking
9           FlightRequest frRequest = initBFRequest(tbRequest);
10          FlightResponse flight = bookFlight(frRequest);

            // Car booking
11          CarRequest crRequest = initBCRequest(tbRequest);
12          CarResponse car = bookCar(crRequest);

            // Confirmation
13          String msg;
14          if (hotel.isBooked() && flight.isBooked() && car.isBooked())
15              msg = "Travel booked. Confirmation no.: " + generateNum();
16          else
17              msg = "Your travel could not be booked";
18          replyBooking.setInformation(msg);
19      }
20      else {
21          replyBooking.setInformation("Invalid credit card");
22      }

23      return replyBooking;
24  }
```

(a)

```
public interface TravelBookingItf {
    @Service(name = "TravelBooking", namespace = "...", port = "...")
    CCardResponse checkCreditCard(CCardRequest ccRequest);

    @Service(name = "TravelBooking", namespace = "...", port = "...")
    HotelResponse bookHotel(HotelRequest hrRequest);

    @Service(name = "TravelBooking", namespace = "...", port = "...")
    FlightResponse bookFlight(FlightRequest frRequest);

    @Service(name = "TravelBooking", namespace = "...", port = "...")
    CarResponse bookCar(CarRequest crRequest);
}
```

(b)

Figure 6.5: Java StarSs version of the travel booking composite service: (a) main program of the composite and (b) task selection interface. In (a), the calls to external services are underlined.

6.4.2 Programming with Objects

ProActive [99] is an object-oriented parallel programming model and runtime for general distributed-memory infrastructures (clusters, grids, clouds). ProActive applications achieve parallelism by creating 'Active Objects' (AOs), which can be deployed remotely and run concurrently. Each AO has its own thread of control and serves incoming (and possibly remote) requests for the execution of methods. Method calls on AOs can be asynchronous, returning future objects that force synchronisation when accessed (a mechanism that is called 'wait by necessity'). ProActive also implements 'automatic continuation', which allows future objects to be passed as parameters of calls on AOs, synchronising when the data is actually accessed in the method body.

ProActive shares the same underlying computational model as Java StarSs, based on spawning asynchronous computations as the main program executes. However, they expose that model to the programmer differently. The next subsections go through the basics of how objects are handled in both models. The comparison will be illustrated with a simple version of an application which solves the classical N-body problem [91]. This problem simulates the evolution of a system of N bodies in space, where the position of a body changes depending on the gravitational force exerted by the rest of the bodies.

Figure 6.6 shows the N-body application for Java StarSs, consisting of (a) the main program, written in sequential Java, and (b) the task selection interface declaring two methods to be run as tasks. The universe is divided in domains, each one containing a planet. For every iteration, the force experimented by each planet due to the planets in the rest of domains is calculated; after that, each planet is moved according to that force. For space reasons, the whole code of the ProActive N-body is not provided; nevertheless, Figure 6.7 compares some relevant fragments of both implementations to make the next explanations easier to follow.

6.4.2.1 Deployment

ProActive programmers must manage the application deployment from the source code (lines 1-6 in Figure 6.7, right column). In ProActive, a 'virtual node' is mapped to one or more physical nodes. This mapping is defined inside XML descriptors, loaded from the application to start the deployment of the Java virtual machines in the nodes (lines 1-3). In the example, the virtual node 'Workers' is used to obtain the list of physical nodes (lines 4-6). The launched JVMs will host the AOs created later on in the application. On the contrary, Java StarSs programs do not include any deployment details; instead, at execution time, objects are transferred to nodes by the runtime according to the scheduling of tasks, just like any other type of data.

6.4.2.2 Object Creation

In ProActive, regular object creation is replaced by a library call for those objects intended to be active. i.e. the domains in N-body (line 8, right column).

```
Domain[] domains = new Domain[numBodies];
for (int i = 0; i < numBodies; i++)
    domains[i] = new Domain(new Planet(universe));

for (int iter = 0; iter < numIter; iter++) {
    for (Domain d : domains)
        for (Domain e : domains)
            if (d != e) d.addForce(e);

    for (Domain d : domains)
        d.moveBody();
}

for (Domain d : domains)
    d.getPlanet().print();
```

(a)

```
public interface NBodyItf {
    @Method(declaringClass = "nbody.Domain")
    void addForce(Domain d);

    @Method(declaringClass = "nbody.Domain")
    void moveBody();
}
```

(b)

Figure 6.6: Java StarSs version of N-body: (a) main program and (b) task selection interface.

Each call to newActive will trigger the creation of an AO in a given node, i.e. an object plus a thread to serve requests on that object. In Java StarSs, objects are created in a regular way and no API call is needed (line 1, left column); moreover, those objects are just data, they do not have any associated thread.

6.4.2.3 Asynchronous Computations

In both programming models, asynchronous computations are spawned as a result of method calls. However, the execution model behind them is different.

Regarding ProActive, the asynchronous invocations are performed on objects created as active. Each computation is, then, linked to a certain AO and will be served in the node where that object and its thread reside. Asynchronous calls can happen between the main program and an AO or between any pair of AOs - in the ProActive N-body, each domain AO calls addForce on the rest of domains - but the programmer must manage the AO references and ensure that no deadlock takes place.

Concerning Java StarSs, the methods that will spawn asynchronous tasks are only those declared in the task selection interface (Figure 6.6(b), methods addForce and moveBody of class Domain). The node where those methods will finally run is not determined by the callee object - in fact, several nodes can have a copy of that object - but by the task scheduling algorithm.

Sequential Java (used by Java StarSs)		ProActive
	deployment	1 GCMApplication gcmad = 2 PAGCMDeployment.loadApplicationDescriptor(**new** File(xmlF)); 3 gcmad.startDeployment(); 4 GCMVirtualNode workers = gcmad.getVirtualNode("Workers"); 5 workers.waitReady(); 6 Node[] nodes = workers.getCurrentNodes().toArray(**new** Node[]{});
1 domains[i] = **new** Domain(**new** Planet(universe));	object creation	7 Object[] pr = **new** Object[] { **new** Planet(universe) }; 8 domains[i] = PAActiveObject.newActive(Domain.**class**,pr,nodes[i]);
2 d.addForce(e);	async co.	9 d.addForce(planet);
3 d.getPlanet().print();	synchro	10 d.getPlanet().print();
	termination	11 **for** (Domain d : domains) 12 PAActiveObject.terminateActiveObject(d, **false**);

Figure 6.7: Comparison of key fragments in the N-body application.

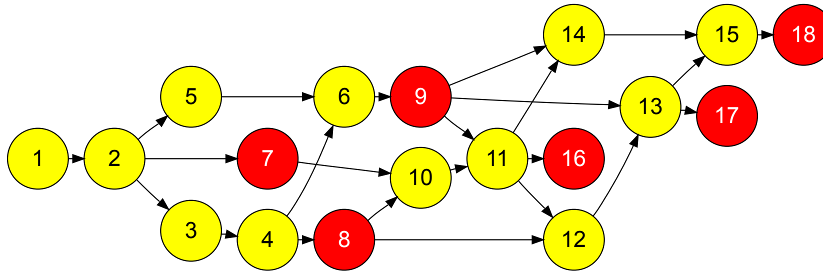


Figure 6.8: Task dependency graph generated for N-body, with a universe of 3 domains and 3 iterations. Yellow (light) tasks correspond to the addForce method, whereas red (dark) ones represent calls to moveBody.

6.4.2.4 Synchronisation

Both programming models support futures, i.e. objects returned by an asynchronous computation. In the example, for the ProActive version, the call d.getPlanet() returns a future object of class Planet, which is immediately accessed by invoking print() on it. This triggers a synchronisation for the result of the getPlanet call.

In addition to returned objects, the Java StarSs runtime automatically synchronises the accesses to any object that participates in a task (i.e. also callees and parameters). Moreover, such synchronisation can take place between two tasks or between a task and the main program. The main loop in Figure 6.6(a) generates moveBody and addForce tasks, which update the domains. The synchronisation between such tasks is enforced by the task dependency graph built on the fly by the runtime (see Figure 6.8). At the end of the application, the invocations of getPlanet on the domain objects need to be done on the last version of each domain, produced by the tasks. Hence, before getPlanet is called on a given domain, the runtime blocks the application thread until the right (last) version of that domain is obtained.

In ProActive, that kind of data dependencies would have to be managed manually in the application code. For instance, if an AO call modified an object parameter and this value were required by a subsequent call to another AO, there would have to be a explicit synchronisation and transfer of that value between AOs.

6.4.2.5 Termination

ProActive requires to explicitly terminate an AO and its associated thread (lines 11-12, right column). Oppositely, the Java StarSs runtime automatically takes care of cleaning the objects transferred to the worker nodes during execution.

6.5 Experiments

This section presents a set of experiments carried out in clouds, both private and public. The section starts with the description of the application and the testbed used in the experiments. After that, a first series of tests demonstrate the virtual resource management and elasticity capabilities of the Java StarSs runtime. Finally, a second series of tests show some performance results.

6.5.1 Gene Detection Composite

The experiments in this section will be executing a real example of an e-Science composite service programmed with Java StarSs.

The original application on which the composite is based is a gene detection code [159] designed by members of the Life Sciences department of the Barcelona Supercomputing Center [9]. Its core algorithm is GeneWise [90], a program for identifying genes in a genomic DNA sequence. First, the application finds a set of relevant regions in a DNA sequence, and then runs GeneWise only for those regions, which is faster than scanning the whole DNA. The application is a sequential Perl code that invokes a set of publicly available bioinformatics services by means of SOAP WS libraries. Those invocations are synchronous and, consequently, no parallelism is achieved between service calls.

Such application was ported to Java following the steps of the Java StarSs programming model: first, in a task selection interface, a total of five service tasks and seven method tasks were declared, for them to be the building blocks of the composite; second, following the example of the original Perl, the composite was programmed as a sequential code that invokes the selected tasks.

The structure of the resulting composite is represented on the right side of Figure 6.9. Each box corresponds to a different part of the composite that contributes to the overall process; the task calls in each part generate a fragment of the whole dependency graph, shown inside the boxes. The following points summarise the structure:

- *Genome DB formatting*: translation of the input genomic DNA sequence to two different formats that will be required later in the program.
- *Sequences retrieval*: obtention of a list of amino acid sequences (proteins) that are similar to a reference input sequence.
- *Gene search*: search of the relevant genomic regions of the DNA sequence for each protein.
- *GeneWise*: execution of the GeneWise algorithm for all the relevant regions found.

The leftmost side of Figure 6.9 contains a snippet of the composite code where a couple of task calls are highlighted: first, a call to runNCBIBlastp, a method task, executes the BLAST program [80] to produce a report object that contains all the proteins similar to the reference protein fastaSeq; second, the

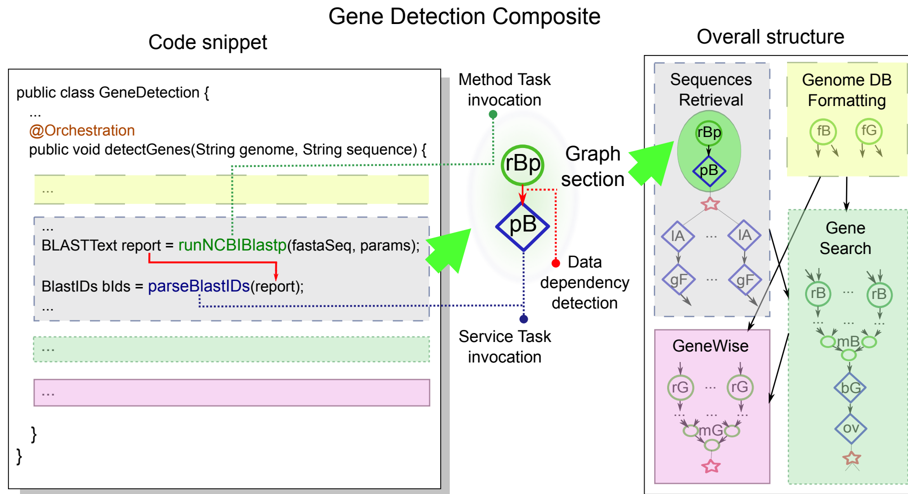


Figure 6.9: Gene detection composite service. The dependency graph of the whole orchestration is depicted on the right of the figure: circles correspond to method tasks and diamonds map to service task invocations, while stars represent synchronisations due to accesses on task result values from the main program. A snippet of the composite code is provided, focusing on a particular fragment which runs BLAST to obtain a list of sequences and then parses their identifiers. The graph section generated by this piece of code is also highlighted in the overall structure of the composite.

parseBlastIDs service task takes that report as input and parses the identifiers of those proteins. Notice how the data dependency between the two tasks is automatically detected. The whole code of the composite and the corresponding task selection interface can be found in Appendix A.3.

It is worth pointing out that, although both the Java StarSs and the original versions are programmed sequentially, they behave differently at execution time: while the Perl script runs serially, Java StarSs asynchronously generates a graph to exploit the parallelism between tasks. Furthermore, service tasks and method tasks can be easily combined, like in the case of runNCBIBlastp and parseBlastIDs: they are called and exchange data just like in a regular Java program.

6.5.2 Testbed

The testbed used in the experiments is formed by the following actors and infrastructures (see Figure 6.10):

- *Client*: Java application that invokes the gene detection composite service.
- *Composite server*: machine running an Apache Tomcat 7.0 WS container [8] that hosts the gene detection service. It is a dual-core Intel

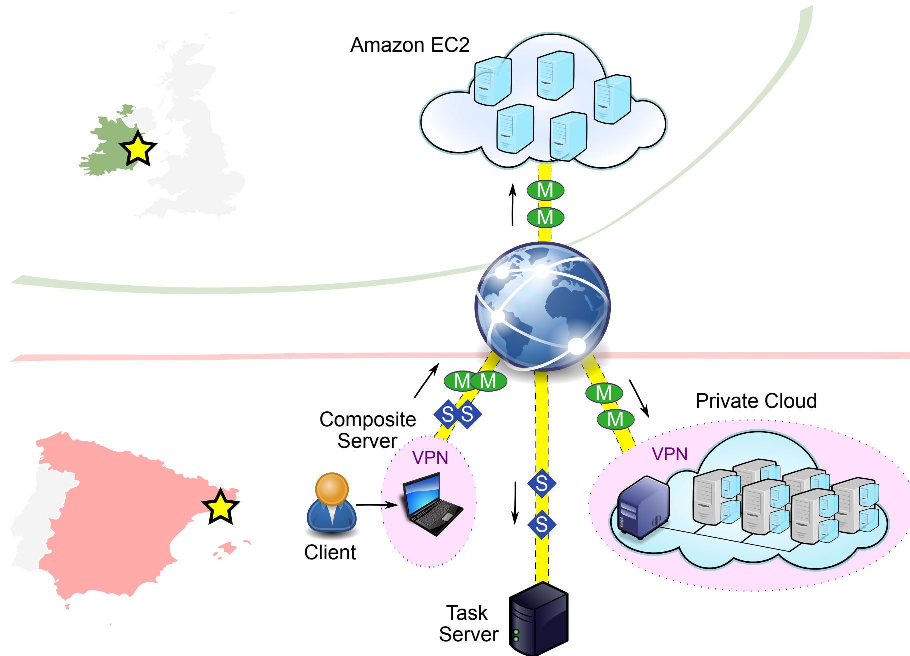


Figure 6.10: Testbed comprising two clouds: a private cloud, located at BSC, and the Amazon EC2 public cloud (Ireland data centre). The GeneDetection composite service is deployed in a server machine, which contacts the VMs of the private cloud through a VPN. An external server publishes the operations corresponding to service tasks.

Core i7 at 2.8 GHz, 8 GB of RAM and 120 GB of disk space. Both the composite's main program and the Java StarSs master runtime execute in this machine. This machine also runs an OpenVPN [45] client.

- *Task server*: machine running an Apache Tomcat 7.0 WS container, which hosts a service that offers the service task operations. Such container is contacted by the Java StarSs runtime to execute service tasks called from the composite.

- *Private cloud*: cluster managed by EMOTIVE Cloud as an IaaS virtualisation layer. On the one hand, the cluster has a front-end node that acts an OpenVPN server and EMOTIVE scheduler. On the other hand, a total of 7 nodes are used for hosting VMs: 3 nodes with two eight-core AMD Opteron 6140 at 2.6 GHz processors, 32 GB of memory and 2 TB of storage each; 4 nodes with two six-core Intel Xeon X5650 at 2.67 GHz processors, 24 GB of memory and 2 TB of storage each. The nodes are interconnected by a Gigabit Ethernet network. The Client, Composite

server, Task server and Private cloud are all located in the BSC/UPC premises in Barcelona, Spain.

- *Amazon EC2*: public IaaS Cloud provider. In the tests, all the Amazon VMs are deployed in the European Union West zone, which corresponds to a data centre located near Dublin, Ireland.

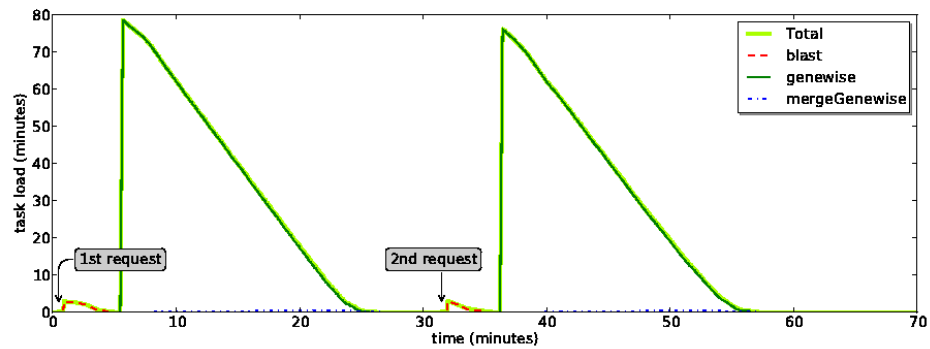
A typical execution begins when a Client issues a WS invocation request to the gene detection service published in the Composite server. This triggers the execution of the composite, leading to the creation of new method and service tasks. The Java StarSs runtime executes service tasks by issuing WS requests to the Task server container. Method tasks are run in VMs on the Private cloud or on Amazon EC2. In the case of the Private cloud, the Composite server and the VMs belong to the same virtual private network, so that they can communicate through SSH. Regarding Amazon, the VMs are also contacted by SSH to their public IP addresses. All the VMs run a Linux distribution where the Java StarSs worker runtime, BLAST and GeneWise have been pre-installed.

6.5.3 Resource Elasticity and Cloud Bursting

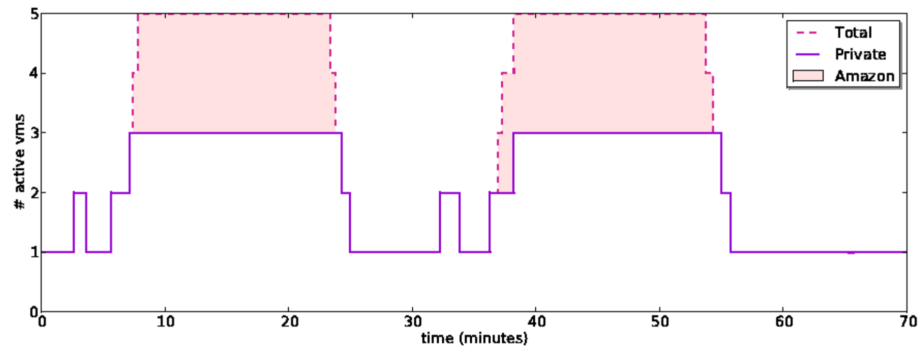
This subsection presents some experiments to demonstrate the elasticity capabilities of the Java StarSs runtime in Cloud environments. In the tests, not only private cloud resources will be reserved as the base infrastructure, but also there will be bursting to a public cloud to meet peak demands. The advantage of such a hybrid deployment is that one only pays for extra compute resources when they are needed.

Figure 6.11 aims to illustrate how elasticity works in the Java StarSs runtime. The figure depicts the arrival of two requests for the gene detection service at the Composite server and their corresponding executions. In (a) the load generated by the executions of the composite can be found - in particular that of the method tasks, which are the ones executed in VMs under the control of the runtime. The instants when each of the two requests arrive are indicated in the figure. Both requests have the same parameters, and thus they generate equivalent loads. The plotted load corresponds to the estimated execution time (in minutes) of all the dependency-free tasks that the runtime is processing at a given moment. Although there are seven types of method task, only the three that are most relevant in the overall load are shown: blast and genewise run those two bioinformatics tools, while mergeGenewise merges intermediate GeneWise results. It can be observed how GeneWise is responsible for most of the computation of the composite.

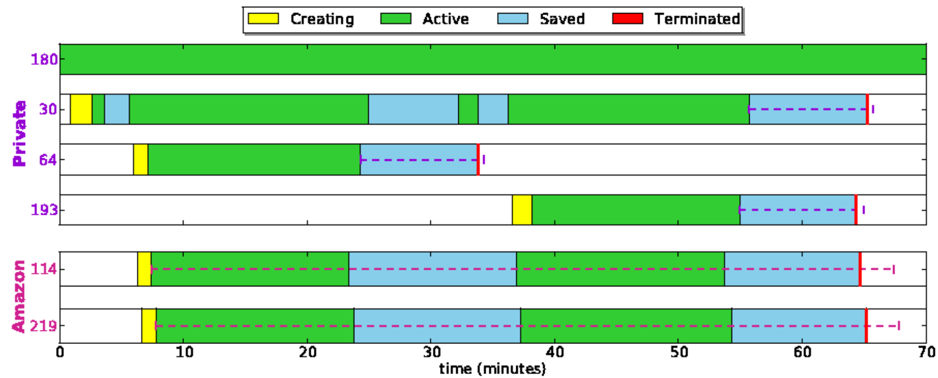
Figure 6.11(b) plots the evolution of the number of VMs for both providers (Private and Amazon). Here, the maximum number of Private and Amazon VMs was set to three and two, respectively. In both providers, the number of cores per VM requested is one (more precisely, in Amazon the instance type is 'm1.medium'). As a complement, Figure 6.11(c) represents the state of the VMs during the considered time interval. The possible VM states are: *Creating*, if the VM has been requested to the provider and is being created and booted;



(a)



(b)



(c)

Figure 6.11: Execution of two requests for the gene detection composite that illustrates the elasticity and bursting features of the Java StarSs runtime: (a) evolution of the load generated by the composite's method tasks; (b) evolution of the number of VMs in the private cloud and Amazon EC2; (c) state of the VMs during the execution of the requests.

Active, if the VM is ready to be used by the task scheduler; *Saved*, if the VM is no longer necessary given the current load, but it is saved for later reuse; *Terminated*, if the provider has been contacted to delete the VM.

When the first request is received, there already exists a Private VM that was created when the service container started; this VM constitutes the resource critical set and it will always be kept active. This is the sequence of relevant events in the depicted time interval:

- *First VM creation*: the first request initially spawns a set of blast tasks, which produce a small increase in the load. As a response, the Java StarSs runtime asks for a new VM (30); the chosen provider is Private, since its VMs are cheaper than Amazon's. When a VM is created, a key set is configured for SSH access and the classes that implement the method tasks are deployed on that VM.

- *First VM saving*: soon, the load decreases again and VM 30 is not needed anymore: at this point (minute 4) VM 30 is saved. Saving a VM means that the connector puts that VM aside so that the scheduler does not take it into account, but it is not destroyed just in case it is needed again in the near future.

- *Bursting*: around minute 5 the appearance of genewise tasks causes a sudden and huge increase in the total task load. Such increase cannot be handled with Private VMs alone, which are limited to three, and hence the Java StarSs runtime relies on Cloud bursting: after Private VM 64, two Amazon VMs are also requested (114 and 219). Note how VM 30 will also help with this load increase, but it does not have to be created: it was saved and now it can immediately become active again.

- *Deadline of a Private VM*: around minute 23, the genewise load is almost gone, which makes the Java StarSs runtime progressively save VMs (starting with the Amazon ones, the most expensive). In the case of Private VMs, they are saved for only up to 10 minutes, and after that time the provider is requested to terminate them (the dashed lines in the VM bars of Figure 6.11(c) represent the VM deadlines). This happens with VM 64 in minute 33: the next spike in load does not arrive soon enough, and the connector decides to turn it off. Before deleting a VM the runtime saves its critical files, that is, those only present in that VM and needed by at least one of the current tasks.

- *Reuse of VMs*: the genewise tasks coming from the second request lead to another peak load in minute 36; conveniently, three VMs are still saved (30, 114, 219) and can be reused, but since VM 64 was deleted a new Private VM (193) is created to take its place.

- *Final VM deadlines*: once the second bag of genewise tasks is completed, a total of four VMs are saved (around minute 55). As explained above, Private VMs are deleted when 10 minutes have passed since they were

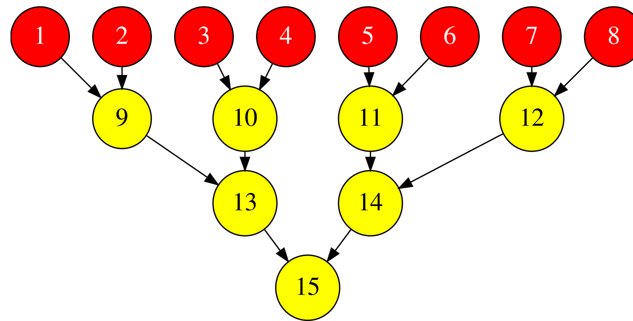


Figure 6.12: Graph generated by the GeneWise computation in the gene detection composite, for an execution that finds 8 relevant regions in the genomic sequence. Red (dark) tasks correspond to the genewise method, whereas yellow (light) ones represent calls to mergeGenewise.

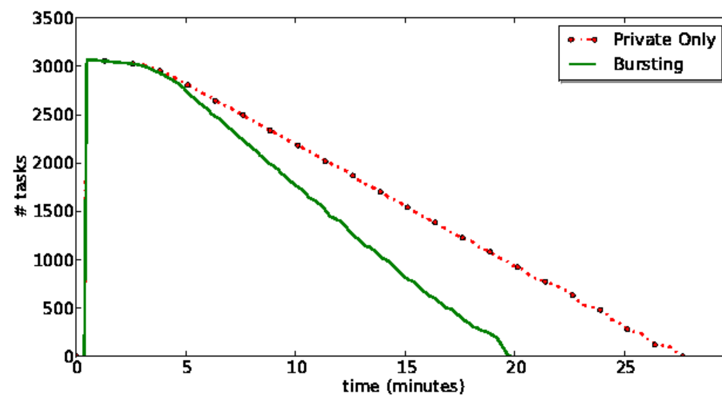
saved. Regarding Amazon VMs, the termination policy is different: Amazon rents its VMs in slots of one hour, i.e. when creating an Amazon VM a full hour is paid in advance, even if the VM is terminated before that hour ends. Consequently, the Amazon EC2 connector of Java StarSs tries to make the most of a VM and only terminates it when the end of its hour slot is approaching and it is in 'Saved' state (the dashed lines in Amazon VM bars represent the duration of the hour slot).

6.5.4 Performance

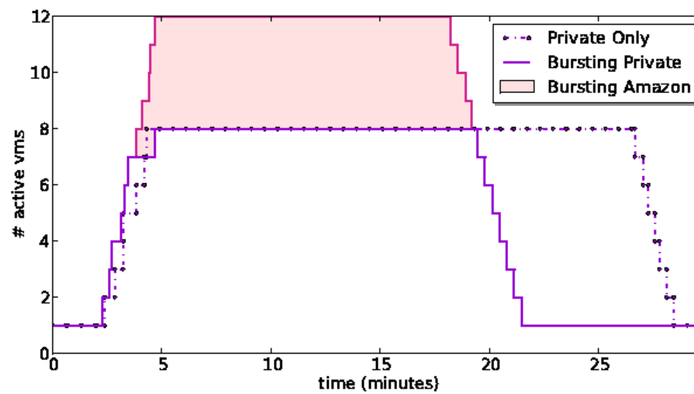
This subsection gives some performance and scalability results of executing the gene detection composite. In particular, the tests will focus in the most computationally-intensive part of the composite, that is, the execution of the GeneWise algorithm on a set of relevant regions of the genomic sequence. This part of the application generates a graph with the shape of a reversed binary tree, like the one in Figure 6.12, which first runs GeneWise on every relevant region previously found and then merges all the partial reports into one.

As a continuation of Section 6.5.3, which discussed elasticity and bursting, the following tests measure how long it takes to execute the GeneWise computation with two configurations: first, only acquiring VMs from the private cloud; second, reserving Amazon VMs as well. In the experiments, the Private VMs have 4 cores, 2 GB of RAM and 1 GB of storage (home directory). The Amazon VMs are of type 'm1.xlarge' (extra large), which also features 4 cores, 15 GB of RAM and 1690 GB of storage. For the results to be more consistent, the EMOTIVE Cloud scheduler was configured to create Private VMs solely in the AMD machines of the private cloud, since the performance of a Private VM differs slightly depending on where it is launched (AMD node or Intel node).

The two lines in Figure 6.13(a) plot the sum of genewise and mergeGenewise tasks that are either running or waiting to be scheduled. The configuration



(a)



(b)

Figure 6.13: Execution of the GeneWise computation, with private VMs only and bursting to Amazon: (a) evolution of the number of tasks, (b) VM elasticity.

called 'Private Only' sets a maximum number of 8 Private VMs (no Amazon VMs are allowed). In the 'Bursting' configuration, in addition to the 8 Private VMs, up to 4 Amazon VMs can be reserved. The cost of an extra large Amazon VM in the EU West zone is \$0.68 per hour plus tax. In both configurations, the execution starts from 1 Private VM and then new VMs are progressively acquired from the corresponding providers - the evolution of the number of machines is depicted in Figure 6.13(b). To complement the figure, Table 6.1 contains information about the tasks for each configuration. Times are calculated by getting a time stamp right before submitting the task and right after a task end notification is received. The real computation is in the genewise tasks, whose duration can vary significantly depending on the length of the genomic region to explore, while the mergeGenewise tasks are more light-weight. On

Table 6.1: Statistics of the GeneWise part of the gene detection composite. Times in seconds.

Task name	**Private Only**		**Bursting**	
	genewise	**mergeGW**	**genewise**	**mergeGW**
# tasks	3068	3067	3068	3067
Avg time	12.62	0.20	12.07	0.21
Min time	2.42	0.18	2.77	0.18
Max time	45.49	4.93	44.36	2.93

the other hand, in 'Bursting' the average task times are a little lower because the Amazon VMs are a bit more performing than the Private ones, which also contributes to decrease the overall execution time.

As can be seen in Figure 6.13, the fact of outsourcing some of the tasks to Amazon EC2 helps finish the GeneWise computation in 8 minutes less, which means a reduction of about 29% in execution time with 33% more cores. In this case, there are at least a couple of factors that prevent the runtime from achieving better results. First, the elastic resource manager/scheduler of Cloud Java StarSs is not locality-aware: it applies a round-robin scheduling algorithm on the available resources, and re-balances the task load when a VM is obtained/removed; this increases the number of transfers to be performed. Second, when sending computations and data to a distant infrastructure like the Amazon data centre, the communications suffer from some extra latency. These factors will be further discussed in the next experiments.

Thus, a second series of tests provides some scalability results for the execution of GeneWise with private and public VMs. In this case, the VMs are created beforehand, and so there is no delay associated with progressively acquiring/releasing VMs. Furthermore, the scheduling algorithm applied does take into account data locality in order to reduce the number of transfers. The VMs have the same characteristics as in the previous tests (4 cores each).

Figure 6.14 depicts the execution times (in logarithmic scale) of the GeneWise computation for different numbers of cores, more precisely the average of three executions per number of cores. The number of tasks per execution and their granularity are equivalent to those specified in Table 6.1. One line corresponds to runs with only private VMs ('Private'), while the other line plots the combination of both Private and Amazon VMs ('Hybrid').

The measures show that Java StarSs achieves good scalability, especially when running the whole computation in one Cloud provider ('Private'). In the 'Hybrid' executions, the results are affected by the distributed nature of the testbed (Figure 6.10). When distributing the tasks of the GeneWise computation graph over more than one provider, task dependencies eventually lead to data transfers between VMs in different providers, even if the locality-aware scheduling algorithm of Java StarSs tries to minimise the number of transfers. Providers can be geographically dispersed, like in our testbed, and consequently latencies become similar to those of the Grid scenario. Moreover, there can be no connectivity between VMs of different providers: this happens in our case,

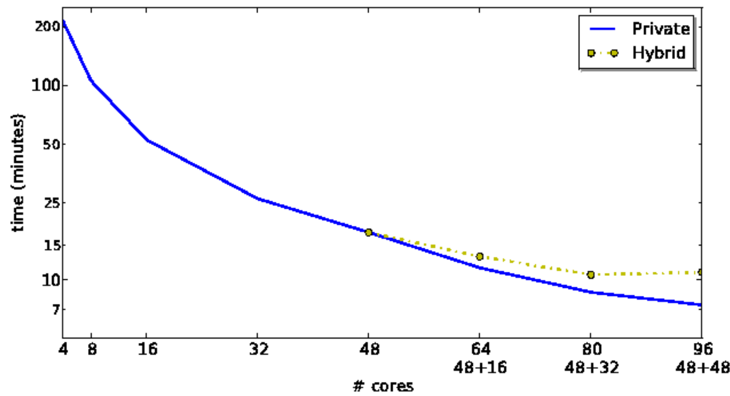


Figure 6.14: Execution times of the GeneWise computation, with private VMs only ('Private') and a combination of private and public VMs ('Hybrid').

where every data transfer between a Private and an Amazon VM passes through the Composite server first. However, the fact that the GeneWise reports are rather small (up to a few kilobytes) helps scale better the application even under those conditions.

On the other hand, task granularity (see Table 6.1) also plays an important role in the performance of GeneWise. The variability in the duration of the genewise tasks challenges the load balancing mechanism of the runtime, while the small execution time of mergeGenewise complicates the overlapping of transfers and computation, particularly for the 'Hybrid' case where the average transfer time is higher. Nevertheless, even if they are hardly worth distributing, the mergeGenewise method invocations are run as tasks to prevent the main program from having to reduce all the partial GeneWise reports.

As a final note, in order to improve the hybrid private-public scenario, Java StarSs could link entire composite runs to specific providers. In a server that receives multiple requests for composites, this could be done by bursting to a public cloud the whole execution of a composite, instead of offloading some of the tasks of a composite that is already being executed in private VMs. This strategy would execute full graphs in VMs of a sole provider, so that data dependencies and their associated transfers always happen inside the boundaries of a provider. The elasticity mechanism of the Java StarSs runtime would decide to create a VM in Amazon only under two conditions: first, a new request for a composite arrives at the server and, second, the private VMs are already overloaded; from that point on, all the tasks generated by that new execution of the composite would be scheduled in public VMs. Hence, setting affinity between composite runs and providers would be especially useful when the size of the data exchanged by tasks is big and transfers through the Internet are more costly. A runtime that applies such a strategy is left as future work.

6.6 Related Work

The service-oriented nature of Cloud computing and its convergence with SOA brings a need for, on the one hand, programming models that ease the development of applications composed by services and, on the other, systems which orchestrate the execution of those services in the Cloud.

In that sense, Java StarSs contributes with a model for true Cloud-unaware programming and easy service composition, combined with a runtime for automatic service orchestration.

This section overviews other approaches in the area of Cloud and service-oriented programming and highlights the differences with respect to Java StarSs.

6.6.1 Platform-as-a-Service Solutions

The Java StarSs programming model and runtime are linked to the concept of PaaS, since they offer means to program and run applications in the Cloud (see Section 6.1.1). Numerous PaaS solutions have appeared so far, providing APIs, tools and infrastructure abstractions to build up network-accessible software - most commonly web sites and web applications. The market is dominated by public PaaS, each managed by a company that restricts the execution of the developed applications to that company's data centres.

One example of PaaS is Microsoft Azure [37]. Azure is primarily based on Microsoft technologies (e.g. Azure VMs can only run a Windows operating system). The programming model features two roles in which the application code can be structured: first, Web role, representing a web server that accepts and processes HTTP requests; second, Worker role, which is typically used for background processing tasks. Each instance of these roles executes in a VM on the Azure platform. Two roles can communicate by means of queues, and they can access persistent storage through the 'blobs' and 'tables' APIs.

Google App Engine [20] is a platform for programming, testing and deploying web applications. It features three runtime environments for Java, Python and Go (Google's language) and provides APIs to e.g. access a database (known as 'datastore'), invoke external services or queue tasks for background execution. App Engine places some restrictions on what the applications can do, either for the sake of performance and scalability (e.g. join operations on the database are not allowed) or for security and isolation reasons (applications run in a sandbox environment where they cannot write to local files nor open sockets).

There exist some differences between Java StarSs and the aforementioned PaaS. First, Java StarSs is not tied to a particular infrastructure: it can potentially operate on top of any Cloud provider thanks to the connector-based design of its runtime, explained in Section 6.2, thus preventing vendor lock-in; there is some work in progress on providing Java StarSs with a development environment, analogous to what other PaaS offer, which will assist the programming steps with the model and also automate the deployment of the application in the desired provider. Second, Java StarSs programmers do not need to invoke any API from the application code; instead, aspects like external service in-

vocation, message exchange or data transfer and synchronisation are handled transparently by the runtime. Third, Java StarSs is not restrictive regarding what a Java programmer can do in her code: she can freely use any Java data type or standard Java library. Finally, there is no particular structure to which the user must adhere when programming; a Java StarSs application can follow any pattern and generate any arbitrary graph of tasks, while the runtime takes care of the data dependencies.

6.6.2 Frameworks for Service Composition

Several frameworks have been proposed to combine services in a process-oriented way. Such frameworks are normally formed by three elements: first, a composition language or model that defines how to specify the services involved in the composition and their relationships; second, a development environment that is commonly a visual editor to add and edit services; third, a runtime system that executes the process logic by orchestrating the composite service.

The most prominent example in this area is the WS-BPEL language [40] standard, already discussed in Section 6.4.1, together with any editor and workflow engine that supports service composition and orchestration based on WS-BPEL (e.g. Eclipse BPEL Designer [12] and Apache Orchestration Director Engine [7], respectively). In contrast to this kind of approaches, the workflow of a Java StarSs application (i.e. the task dependency graph) is not defined graphically, but dynamically generated by the logic of the main program itself as it runs: each invocation of a selected method or service is replaced on-the-fly by the creation of an asynchronous task which is added to the graph. Furthermore, graphical editors require the user to manually specify the dependencies (links) between the services of the composition, while Java StarSs discovers those dependencies automatically.

Also regarding the composition of service-based processes, there exists the concept of 'mashup' [89]: a web application created by combining existing web resources (services, APIs, data sources), possibly belonging to different domains, in order to build a new service with an added value. Most of those resources are accessed as RESTful services (REpresentational State Transfer [114]); such services are designed to use basic HTTP operations (GET, POST, DELETE, PUT) as methods and they represent a more light-weight alternative to SOAP Web services. There are many examples of mashups, like mapping mashups (e.g. ChicagoCrime.org [15] combines data from the Chicago Police Department's online database with cartography from Google Maps) or search and shopping mashups (e.g. Bizrate [10] aggregates comparative price data obtained from different vendors). Besides, platforms like Yahoo Pipes [72] or FAST [179] allow non-skilled users to visually construct mashups out of pre-built gadgets.

Like in the case of WS-BPEL editors, Java StarSs differs from the aforementioned mashup platforms in the way services are composed: programmatically versus graphically. On the other hand, Java StarSs does not currently support RESTful services as tasks - only SOAP-based ones. Nevertheless, it could be easily extended at two levels to support them: first, concerning the program-

ming model, a method representing a RESTful service could be declared in the task selection interface, along with specific Java annotations to define its attributes (e.g. URI of the target resource and HTTP operation); second, the runtime could invoke the external service through a REST client API like the one provided by Jersey [36].

JOLIE [143] is analogous to Java StarSs in the sense that it permits to textually program service compositions. However, JOLIE features a custom language with a certain syntax to write the main program and invoke services from it, while Java StarSs relies on a widely-known language like Java. Furthermore, JOLIE requires the user to explicitly deal with parallelism (by specifying operators between statements) and data dependencies (by binding input and output ports of services), whereas Java StarSs relies on sequential Java programming and leaves all that burden to the runtime.

6.6.3 Cloud Programming Models

This subsection analyses some approaches that, like Java StarSs, have evolved from the Grid/Cluster scenario into a more Cloud-oriented perspective.

MapReduce [103] is a programming model and software framework for writing applications that process vast amounts of data in parallel. In the MapReduce model, the application code is basically divided in a 'map' function that processes a key/value pair to generate a set of intermediate key/value pairs, and a 'reduce' function that merges all the intermediate values associated with the same intermediate key. MapReduce applications have shown high scalability when running in large commodity clusters and, with the popularisation of Cloud computing, some platforms have started to facilitate the execution of MapReduce programs in Cloud infrastructures too, like Amazon Elastic MapReduce [3] and Google App Engine MapReduce [21]; the process usually involves uploading the input data to the vendor's storage, making the application invoke a supported MapReduce API and establishing limits on the VMs to use. MapReduce is a powerful yet simple model that has gained widespread adoption, but it is only suitable for a set of applications that can be expressed in the 'mappers and reducers' pattern. On the contrary, Java StarSs is more flexible and can accommodate a broader range of applications, which can generate any arbitrary workflow graph.

Aneka [177], originally a .NET-based software system for the creation of enterprise grids, has moved to a market-oriented Cloud PaaS. The most important changes in Aneka concern its runtime and how it manages dynamic provisioning and accounting of virtual resources in private and public clouds. Regarding the programming model, Aneka provides a software development kit to write three types of application: first, task-based, for expressing bags of independent tasks; second, thread-based, for porting multi-threaded applications to a distributed environment; third, MapReduce applications. None of these alternatives allow to create workflows with automatically-controlled dependencies nor address the easy development of composite services for the Cloud, which are two key characteristics of Java StarSs.

ProActive, already discussed in Section 6.4.2, offers a new resource manager that, like the Java StarSs runtime, can use Grid resources and eventually burst to Amazon if necessary [81]. However, the ProActive programming model has not been extended for use in Cloud environments and it lacks proper service-orientation: although an active object can be hidden behind a service interface, there is no special support for orchestration of several service active objects.

6.7 Summary

This chapter has provided an overview of the third and last infrastructure contemplated in this dissertation: the Cloud. Such infrastructure is mainly characterised by, on the one hand, a service-oriented approach that delivers different types of technology as services over the Internet and, on the other, virtualised resources that can be elastically acquired and released to scale with demand and that are paid as-you-go.

Due to the service-based nature of the Cloud, Cloud applications can benefit from the principles of the Service-Oriented Architecture style, which defines applications as composite services that combine other services to provide an added value. Those composites, delivered in the form of Software-as-a-Service, need to be orchestrated in the Cloud while exploiting its elastic resource provisioning.

Therefore, Cloud applications require both programming models that ease the development of composite services and systems that steer the execution of those services in the Cloud. As a response, the Java StarSs programming model and runtime were extended to support service invocation, composition and orchestration in Cloud environments, as well as to handle objects - a data type commonly found among the parameters of a service. The Cloud flavour of Java StarSs can be fully integrated in a service platform, where composites are published as service operations and thus they can be invoked by multiple clients; those requests for the execution of composites lead to the generation of multiple task dependency graphs, possibly formed by both method and service tasks, and the Java StarSs runtime manages the execution of those workflows in the Cloud. For that purpose, the runtime is also able to increase and decrease the number of virtual resources depending on the current task load.

As a first step to evaluate Cloud Java StarSs, its programmability has been compared to other approaches regarding two aspects: first, how are composite services created from other services and methods, and how their execution is steered; second, how are objects manipulated in the main program and passed to tasks for them to participate in remote computations. This study demonstrates that Java StarSs allows to easily combine services and methods that access objects in the form of plain-Java sequential programs, while the orchestration of the generated tasks and the management of their data are completely delegated to the runtime.

In a second step of the evaluation, a set of experiments have demonstrated how Java StarSs can orchestrate the execution of a SaaS composite, formed by calls to external services and normal methods as tasks, in a real Cloud setup. On

the one hand, it has been discussed how Java StarSs exploits Cloud elasticity, dynamically increasing/decreasing the number of virtual resources depending on task load; furthermore, such elasticity can combine both private and public clouds (bursting) to face sudden spikes in load. On the other hand, a scalability study has shown good results when running the computationally-intensive part of an e-Science composite in both a private cloud and a combination of private and public virtual machines.

Chapter 7

Conclusions and Future Work

Two fundamental facts have shaken up the computing landscape in the last decade: on the one hand, the increase in complexity and size of new parallel and distributed infrastructures and, on the other, the growing need of some applications for computing and storage resources. In such a scenario, programmers face the challenge of developing applications that exploit those infrastructures, which is often not an easy undertaking. Dealing with duties related to parallelisation and distribution, or with the specifics of a particular infrastructure, complicates such task.

In that sense, there is a strong need for programming models that build a bridge to connect infrastructures and applications. Moreover, these models must target programming productivity, understood as a tradeoff between programmability and performance, which has become crucial for software developers. Therefore, highly-productive models are required to provide simple means for writing parallel and distributed applications; in addition, such applications must be able to run on current infrastructures without sacrificing performance.

This thesis has contributed to address the programming-productivity challenge with **Java StarSs**, which includes (i) a **parallel programming model** for distributed Java applications and (ii) a **runtime system** that implements the features of the model for three different distributed parallel infrastructures. The next subsections go into further detail about these contributions, discussing the conclusions obtained and some envisaged future work.

7.1 Programming Model

In light of how the computing scene has evolved, it seems inevitable that programmers will have to change the way they approach software developing. Since parallelising and distributing a sequential code in an automatic and efficient manner still seems unfeasible, the intervention of the programmer is required

to some extent. However, we argue that *such intervention does not have to include the duties that make parallelisation and distribution hard*, like thread creation and synchronisation, messaging or fault tolerance. Instead, those can be delegated to a runtime system, while the user only provides hints to help with the process. It is advisable, though, that the user has a good knowledge of her application, that is, of the computations that compose that application and the data they access and share, so that she can reason about opportunities for parallelism and structure the application in a manner that eases the work of the runtime.

Such philosophy has been demonstrated with the Java StarSs programming model, by means of its three main characteristics. First, the model is based on *fully-sequential programming*, which eliminates the need for explicitly dealing with parallelisation constructs and libraries in the code and makes the model appealing to those users that lack concurrent programming expertise. Second, the model *abstracts* the application from the underlying distributed infrastructure: Java StarSs programs do not include any detail that could tie them to a particular platform, which makes them portable between infrastructures as diverse as grids, clusters and clouds, as it has been seen in this thesis. Third, the model is based on a *mainstream language* like Java, which facilitates its adoption to a big community of users that can reuse their knowledge of the language; this is a good quality with respect to alternatives that propose new languages, whose learning curve can be steep (e.g. X10, analysed in this thesis).

In short, Java StarSs applications are sequential plain-Java programs that encapsulate computations in tasks, which can be regular methods or service operations. A task can either contain Java code or act as a wrapper of some functionality programmed in another language. The user is mainly responsible for identifying those tasks, as well as for stating how they access their parameters. The way tasks share data is through those parameters, which can be of any type supported in Java (primitives, files, objects, arrays). Concurrency is implicit in the model, based on tasks that are asynchronously spawned at execution time and whose data dependencies are automatically discovered.

This thesis has shown how *the model is general enough to be applied to a variety of applications*, including e-Science programs, HPC benchmarks or business workflows. Furthermore, the comparison with other representative models and languages has demonstrated the *good programmability* of Java StarSs, as a result of freeing the programmer from things like statically determining data dependencies, specifying data distributions with complex syntax or using APIs to create and invoke remote objects. Contrarily, other models provide means to tune applications for every last bit of performance, which often sacrifices programming expressiveness. In the end, there is no perfect approach, it all depends on the type of user that a given model targets. However, it seems clear that a vast majority of programmers have little or no knowledge in parallel programming, which makes implicit models like Java StarSs indispensable for these people to benefit from new parallel and distributed platforms.

7.1.1 Future work

Since its initial design, the programming model has continuously evolved to incorporate new features, often driven by the needs of users and applications, but always preserving its simplicity and its main characteristics.

In that sense, the model could be extended even further. Perhaps the most interesting change would be to allow for nested tasks. Currently, task nesting is inherently supported for the service-oriented scenario, where a Java StarSs composite can include invocations to external services, selected as tasks. Those services, at their turn, can also be composites deployed in another service container. This creates a SOA where services rely on other (nested) services to provide a new functionality.

Such hierarchical creation of tasks could also be supported for method tasks, which now are only spawned from the main program. The model could permit to generate subtasks from inside a method task, so that the main program contains coarse-grain tasks that encompass other tasks with finer granularity. This maps perfectly to distributed infrastructures with many-core nodes, which seem to be the future of computer architecture, since big tasks can be sent to a node and then be decomposed there into smaller subtasks to feed the cores of that node.

7.2 Runtime System

As an implicit parallel programming model, Java StarSs needs a runtime system that enables its features and abstracts it from what is underneath. Hence, this thesis has presented the design of a runtime that *takes care of several duties on behalf of the programmer*, like data dependency analysis, data transfer or task scheduling. Besides, along with the programming model, the runtime system has also evolved during the realisation of this thesis, in order to *deal with the singularities of each infrastructure* on which it has been implemented. For three different scenarios like grids, clusters and clouds, this thesis has shown how the runtime can keep the specifics of each of them transparent to the model.

The work started by focusing on applications that manipulate files and execute in heterogeneous geographically-distributed grids. Here, the diverse alternatives in terms of Grid middleware were covered at runtime level, so that the programmer did not have to change her application depending on the particular grid to be used. However, any programmer should still bear in mind that grids are characterised by significant latencies, middleware overhead and waiting times, and therefore the granularity of the computations should be chosen accordingly for the application to be worth distributing.

When exploring the cluster scenario, we concluded that the technologies that are suitable for grids are not necessarily convenient for a more homogeneous environment like a cluster. As an example, the JavaGAT library brings interoperability between different kinds of Grid middleware, but at the expense of performance; in clusters, an efficient communication layer that can exploit fast networks is more desirable. On the other hand, the master - worker de-

sign of the runtime also had to be revised to make it more scalable, creating persistent workers that can allocate, cache and exchange in-memory data (arrays, objects). Arguably, in a parallel programming model based on sequential coding, the path to scalability always consists in freeing the main thread from as many responsibilities as possible and distributing them among the rest of threads, perhaps until the point in which the main thread is only in charge of executing the main program and generating the first level of tasks.

Finally, Cloud computing was addressed, first by supporting the orchestration of composite services that invoke other services and, second, by being able to interact with different Cloud providers and elastically manage virtualised resources. The Cloud is a more flexible scenario than the Grid or Cluster ones, not only because of its dynamic resource provisioning, but also regarding the easy creation of custom setups in the form of virtual machine images. Nevertheless, the execution of an application over multiple Cloud providers (e.g. in a private-public hybrid configuration) can lead to an overhead and communication latencies similar to those of a grid and, consequently, offloading work to an external provider should be done with care in order to minimise data transfers across provider boundaries.

Overall, one of the lessons learned along this process is that, no matter the infrastructure, *portability of applications and interoperability are always a major concern*. There is typically a plethora of alternatives to implement and execute an application in a certain scenario, and several vendors compete to make their solutions dominate the market. Standards do appear, either 'de facto' or produced by collaborative organisations that develop them, but it is often complicated for them to be widely accepted. This situation, which is likely to keep happening in future scenarios, increases the importance of systems like the Java StarSs runtime that free the user from porting the same application over different platforms.

All the infrastructure chapters (4, 5, 6) have tested the performance of Java StarSs applications as the second aspect of the productivity analysis, providing experiments in real-world infrastructures. However, it is perhaps Chapter 5 the one that has focused the most on HPC, thoroughly evaluating the performance and scalability of a set of benchmarks. These experiments show that Java StarSs can achieve *remarkable performance* in comparison to other state-of-the-art approaches. In particular, the model is especially suitable for codes with complex data dependencies that are hard to control manually. The Java StarSs runtime handles those cases with no burden for the programmer, automatically detecting the dependencies, defining renamings of written data, transferring them if necessary to balance the task load and caching them for later use. On the other hand, communication-intensive applications with little computation constitute a case where Java StarSs still has room for improvement. Since it is based on sequential Java, Java StarSs lacks collective communication operations that exist e.g. in MPI. Such kind of communications, and more precisely reductions, could be useful in applications like Hmmpfam or GeneDetection, seen in Chapters 4 and 6 respectively, to merge a group of partial results.

7.2.1 Future work

Part of the future work could include studying how to handle collective communications in Java StarSs. The most important condition to fulfill would be to keep the simplicity of the model, based on sequential Java with no use of any API in the application. In the case of reduction operations, one possible option would be to mark a given task as a reduction task in the selection interface. In applications like Hmmpfam where a computation phase is followed by a merge phase, the runtime would know that the partial computation results can be merged in any order by the reduction tasks, and it would invoke such a task as soon as two (or more) partial results are available.

As discussed in Section 7.1.1, the programming model could also be extended to allow task nesting. This would require some support from the runtime system as well. Concretely, the master runtime would partially delegate some functionalities to the workers, like bytecode instrumentation or dependency analysis, for them to be able to generate and process the subtasks of a given task. Notice how such delegation would favour scalability, since the master runtime would not be the only one responsible for task processing, sharing this job with the workers. In addition, master-worker communications would be reduced, since a single task invocation from the main program would lead to the creation of multiple computations in a worker.

Concerning object task parameters, another possible modification to the runtime would be to detect dependencies on subobjects, i.e. objects referenced by other objects; for instance, if a task updated an object `F` which is a field of another object `O`, and a subsequent task read `O`, the runtime would find a data dependency between these two tasks. Furthermore, there is some work in progress to support persistent objects that live beyond the execution of an application and can be loaded in subsequent runs.

Bibliography

[1] Alioth. The Computer Language Benchmarks Game. http://shootout.alioth.debian.org/.

[2] Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2/.

[3] Amazon Elastic MapReduce. http://aws.amazon.com/elasticmapreduce/.

[4] Apache Cassandra. http://cassandra.apache.org/.

[5] Apache CXF. http://cxf.apache.org/.

[6] Apache Hadoop. http://hadoop.apache.org/.

[7] Apache ODE (Orchestration Director Engine). http://ode.apache.org/.

[8] Apache Tomcat. http://tomcat.apache.org/.

[9] Barcelona Supercomputing Center. http://www.bsc.es.

[10] Bizrate. http://www.bizrate.com.

[11] Directed Acyclic Graph Manager, a Meta-scheduler for Condor. http://research.cs.wisc.edu/condor/dagman/.

[12] Eclipse BPEL Designer. http://www.eclipse.org/bpel/.

[13] European Bioinformatics Institute. http://www.ebi.ac.uk.

[14] European Grid Infrastructure. http://www.egi.eu.

[15] EveryBlock's Chicago crime. http://www.chicagocrime.org.

[16] Extensible Markup Language (XML). http://www.w3.org/XML/.

[17] FlexiScale. http://www.flexiscale.com.

[18] Force.com Platform. http://www.salesforce.com/platform/.

[19] gLite User Guide.
https://edms.cern.ch/file/722398/1.4/gLite-3-UserGuide.pdf.

[20] Google App Engine. http://code.google.com/appengine/.

[21] Google App Engine MapReduce .
http://code.google.com/p/appengine-mapreduce/.

[22] Google Gmail. http://www.gmail.com.

[23] GridCafé, CERN. http://www.gridcafe.org.

[24] HMMER: biosequence analysis using profile hidden Markov models.
http://hmmer.janelia.org.

[25] Hypertext Transfer Protocol. http://www.w3.org/Protocols/.

[26] Ibergrid 2011 Year Report.
http://www.es-ngi.es/documentos/Ibergrid_report_2011_downloadable.pdf.

[27] IBM General Parallel File System.
http://www-03.ibm.com/systems/software/gpfs/.

[28] IBM Watson, winner of Jeopardy!
http://www-03.ibm.com/innovation/us/watson/.

[29] Iniciativa Nacional Grid. http://www.gridcomputing.pt.

[30] International Business Machines. http://www.ibm.com.

[31] Java annotations.
http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html.

[32] Java Platform, Enterprise Edition (Java EE).
http://www.oracle.com/javaee.

[33] Java programming assistant. http://www.javassist.org.

[34] Java Secure Channel. http://www.jcraft.com/jsch/.

[35] JavaNumerics, Java Grande Forum Numerics Working Group.
http://math.nist.gov/javanumerics/.

[36] Jersey, JAX-RS (JSR 311) Implementation. http://jersey.java.net/.

[37] Microsoft Azure. http://www.microsoft.com/azure/.

[38] NetSuite. http://www.netsuite.com.

[39] Network File System. http://www.ietf.org/rfc/rfc3010.

[40] OASIS Web Services Business Process Execution Language (WS-BPEL). http://www.oasis-open.org/committees/wsbpel/.

[41] OGSA Information Modeling. https://forge.gridforum.org/sf/go/doc13726.

[42] Open Grid Forum. http://www.gridforum.org/.

[43] Open Portable Batch System. http://www.openpbs.org/.

[44] Open Science Grid. http://www.opensciencegrid.org.

[45] Open VPN. http://openvpn.net/.

[46] OpenNebula. http://www.opennebula.org.

[47] Organization for the Advancement of Structured Information Standards (OASIS). https://www.oasis-open.org/.

[48] OSG Document Database. http://osg-docdb.opensciencegrid.org/.

[49] Platform Load Sharing Facility. http://www.platform.com/workload-management/high-performance-computing.

[50] POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995). http://standards.ieee.org/develop/wg/POSIX.html.

[51] Programming Language Popularity. http://www.langpop.com/.

[52] Rackspace Cloud. http://www.rackspace.com.

[53] Salesforce.com. http://www.salesforce.com.

[54] ScalaLife Project Pilot Applications - DISCRETE. http://www.scalalife.eu/applications.

[55] Simple Object Access Protocol. http://www.w3.org/TR/soap/.

[56] Spanish National Grid Initiative. http://www.es-ngi.es/.

[57] Sun Microsystems. JavaBeans. http://java.sun.com/products/javabeans/.

[58] SUPERFAMILY Database. http://supfam.cs.bris.ac.uk.

[59] Terracotta Distributed Shared Objects. http://www.terracotta.org.

[60] The Eclipse Project. http://www.eclipse.org/.

[61] The European Research Network on Foundations, Software Infrastructures and Applications for large scale distributed, GRID and Peer-to-Peer Technologies - CoreGRID. http://www.coregrid.net/.

[62] The GÉANT pan-European data network. http://www.geant.net.

[63] The Secure Shell (SSH) Authentication Protocol. http://www.ietf.org/rfc/rfc4252.

[64] TIOBE Programming Community Index. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

[65] Travel Booking BPEL Example. http://publib.boulder.ibm.com/bpcsamp/scenarios/travelBooking.html.

[66] Uniform Resource Identifiers (URI): Generic Syntax. http://www.ietf.org/rfc/rfc2396.

[67] Virtual Organization Membership Service. http://edg-wp2.web.cern.ch/edg-wp2/security/voms/.

[68] Web Services Architecture - W3C. http://www.w3.org/TR/ws-arch/.

[69] Web Services Description Language. http://www.w3.org/TR/wsdl.

[70] What the 'Private Cloud' really means. http://www.infoworld.com/t/cloud-computing/what-the-private-cloud-really-means-463.

[71] Worldwide LHC Computing Grid. http://lcg.web.cern.ch.

[72] Yahoo Pipes. http://pipes.yahoo.com/pipes/.

[73] CORBA Component Model Specification, version 4.0. http://www.omg.org/technology/documents/formal/components.html, April 2006.

[74] Basic Features of the Grid Component Model (assessed). CoreGRID Deliverable D.PM.04, 2007.

[75] Integration of Clouds and Virtualisation into the European Production Infrastructure. EGI Inspire EU Deliverable 2.6, 2011.

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

[77] B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, and D. Q. Steven. Efficient data transport and replica management for high-performance data-intensive computing. In *in Mass Storage Conference*, 2001.

[78] G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. van Nieuwpoort, A. Reinefeld, F. Schintke, T. Schütt, E. Seidel, and B. Ullmer. The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid. In *Proceedings of the IEEE*, volume 93, pages 534–550, Mar. 2005.

[79] M. Alt, J. Dünnweber, J. Müller, and S. Gorlatch. HOCs: Higher-Order Components for Grids. In V. Getov and T. Kielmann, editors, *Component Models and Systems for Grid Applications*, CoreGRID, pages 157–166. Springer-Verlag, June 2004.

[80] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[81] B. Amedro, F. Baude, D. Caromel, C. Delbé, I. Filali, F. Huet, E. Mathias, and O. Smirnov. An Efficient Framework for Running Applications on Clusters, Grids and Clouds. In *Cloud Computing: Principles, Systems and Applications*. Springer Verlag, 2010.

[82] B. Amedro, V. Bodnartchouk, D. Caromel, C. Delbe, F. Huet, and L. Taboada, Guillermo. Current State of Java for HPC. Technical Report RT-0353, INRIA, 2008.

[83] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[84] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Euro-Par 2009 best papers issue*, 2010.

[85] R. M. Badia, J. Labarta, R. Sirvent, J. M. Pérez, J. M. Cela, and R. Grima. Programming Grid Applications with GRID superscalar. *Journal of GRID Computing*, 1(2):151–170, June 2003.

[86] M. Baker and R. Buyya. Cluster Computing at a Glance. In *High Performance Cluster Computing: Architectures and Systems*, chapter 1, pages 3–47. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999.

[87] A. Basukoski, V. Getov, J. Thiyagalingam, and S. Isaiadis. Component-Based Development Environment for Grid Systems: Design and Implementation. In *CoreGRID Workshop - Making Grids Work*, pages 119–128, 2007.

[88] C. Bell, D. Bonachea, R. Nishtala, and K. A. Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In

Proc. IEEE International Parallel & Distributed Processing Symposium (20th IPDPS'06), Rhodes Island, Greece, Apr. 2006. IEEE Computer Society.

[89] D. Benslimane, S. Dustdar, and A. Sheth. Services Mashups: The New Generation of Web Applications. *IEEE Internet Computing*, 12(5):13–15, 2008.

[90] E. Birney, M. Clamp, and R. Durbin. GeneWise and Genomewise. *Genome Research*, 14(5):988–995, May 2004.

[91] G. Blelloch and G. Narlikar. A practical comparison of n-body algorithms. In *Parallel Algorithms*, Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1997.

[92] R. F. Boisvert, J. Moreira, M. Philippsen, and R. Pozo. Java and Numerical Computing. *Computing in Science and Engineering*, 3(2):18–24, Mar. 1996.

[93] H. L. Bouziane, C. Perez, N. Currle-Linde, and M. Resch. A Software Component-based Description of the SEGL Runtime Architecture. Technical Report TR-0054, Institute on Grid Systems, Tools and Environments, CoreGRID - Network of Excellence, July 2006.

[94] C. I. Bradford L. Chamberlain. Concurrency Oriented Programming in Erlang. `http://www.guug.de/veranstaltungen/ffg2003/papers/ffg2003-armstrong.pdf`, 2003.

[95] C. I. Bradford L. Chamberlain. Multiresolution Languages for Portable yet Efficient Parallel Programming, White paper. `http://chapel.cray.com/papers/DARPA-RFI-Chapel-web.pdf`, 2007.

[96] A. W. Brown, editor. *Component-Based Software Engineering: Selected Papers from the Software Engineering Institute*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[97] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. An Open Component Model and Its Support in Java. In *Component-Based Software Engineering*, pages 7–22, 2004.

[98] M. Cargnelli, G. Alleon, and F. Cappello. OpenWP: Combining annotation language and workflow environments for porting existing applications on grids. In *Proceedings of the 2008 9th IEEE/ACM International Conference on Grid Computing*, GRID '08, pages 176–183, Washington, DC, USA, 2008. IEEE Computer Society.

[99] D. Caromel, W. Klauser, and J. Vayssiere. Towards Seamless Computing and Metacomputing in Java. *Concurrency and Computation: Practice and Experience*, 10(11–13):1043–1061, September-November 1998.

[100] B. Chamberlain, D. Callahan, and H. Zima. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, 2007.

[101] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[102] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, Jan. 1998.

[103] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[104] E. Deelman, G. Singh, M. hui Su, J. Blythe, A. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus: a framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming Journal*, 13:219–237, 2005.

[105] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good. The cost of doing science on the cloud: the montage example. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 50:1–50:12, Piscataway, NJ, USA, 2008. IEEE Press.

[106] N. N. A. S. Division. NAS Parallel Benchmarks. `http://www.nas.nasa.gov/Software/NPB/`.

[107] A. Duran, R. Ferrer, E. Ayguadé, R. M. Badia, and J. Labarta. A proposal to extend the OpenMP tasking model with dependent tasks. *International Journal of Parallel Programming*, 37(3):292–305, 2009.

[108] S. R. Eddy. Profile hidden Markov models. *Bioinformatics*, 14(9):755–763, 1998.

[109] Edward F. Walker and Richard Floyd and Paul Neves. Asynchronous Remote Operation Execution in Distributed Systems. In *10th Intl. Conf. on Distributed Computing Systems (ICDCS-10)*, pages 253–259, May 1990.

[110] T. A. El-Ghazawi, W. W. Carlson, and J. M. Draper. *UPC Language Specifications*, v1.1.1 edition, October 2003.

[111] M. Farreras. *Optimizing programming models for massively parallel computers*. PhD thesis, Universitat Politècnica de Catalunya, 2008. Advisor: Toni Cortes.

[112] M. Farreras and G. Almasi. Asynchronous PGAS runtime for Myrinet networks. *PGAS10: 4th Conference Partitioned Global Address Space Programming Model*, 2010.

[113] M. Farreras, V. Marjanovic, E. Ayguade, and J. Labarta. Gaining asynchrony by using hybrid UPC/SMPSs. *ICS09: 1st Workshop on Asynchrony in the PGAS Programming Model (APGAS) in the 23rd International Conference on Supercomputing*, 2009.

[114] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, 2000.

[115] R. J. O. Figueiredo, P. A. Dinda, and J. A. B. Fortes. Guest editors' introduction: Resource virtualization renaissance. *IEEE Computer*, 38(5):28–31, 2005.

[116] M. P. Forum. MPI: A Message-Passing Interface Standard. Technical report, Knoxville, TN, USA, 1994.

[117] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Int. Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[118] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Int. J. High Perform. Comput. Appl.*, 15(3):200–222, Aug. 2001.

[119] George Almasi, Ganesh Bikshandi, Calin Cascaval, David Cunningham, Gabor Dozsa, Montse Farreras, David P. Grove, Sreedhar B. Kodali, Nathaniel Nystrom, Igor Peshansky, Vijay Saraswat, Sayantan Sur, Olivier Tardieu, Ettore Tiotto. HPC Challenge 2009 Awards Competition: UPC and X10, 2009.

[120] T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, G. V. Laszewski, C. Lee, A. Merzky, H. Rajic, and J. Shalf. SAGA: A Simple API for Grid Applications. High-level application programming on the Grid. In *Computational Methods in Science and Technology*, 2006.

[121] S. Haridi, P. V. Roy, P. Brand, and C. Schulte. Programming Languages for Distributed Applications. *New Generation Computing*, 16(3):223–261, 1998.

[122] M. E. Hellman. An Overview of Public Key Cryptography. *IEEE Communications Society Magazine*, 16:24–32, Nov. 1978.

[123] J. L. Hennessy, D. A. Patterson, and D. Goldberg. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 2002.

[124] IBM. RSCT LAPI Programming Guide. `http://publib.boulder.ibm.com/epubs/pdf/bl5lpg04.pdf`, 1990.

[125] A. Iosup, S. Ostermann, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema. Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing. *IEEE Trans. Parallel Distrib. Syst.*, 22(6):931–945, June 2011.

[126] K. Jiang, O. Thorsen, A. Peters, B. Smith, and C. P. Sosa. An Efficient Parallel Implementation of the Hidden Markov Methods for Genomic Sequence-Search on a Massively Parallel System. *IEEE Transactions on Parallel and Distributed Systems*, 19(1):15–23, 2008.

[127] P. Kacsuk and G. Sipos. Multi-Grid, Multi-User Workflows in the P-GRADE Grid Portal. *Journal of Grid Computing*, 3(3-4):221–238, 2005.

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

[129] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[130] S. Kumar, G. Dozsa, G. Almasi, P. Heidelberger, D. Chen, M. E. Giampapa, M. Blocksome, A. Faraj, J. Parker, J. Ratterman, B. Smith, and C. J. Archer. The deep computing messaging framework: generalized scalable message passing on the blue gene/p supercomputer. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 94–103, New York, NY, USA, 2008. ACM.

[131] K.-K. Lau. *Component-Based Software Development: Case Studies (Series on Component-Based Software Development)*. World Scientific Press, 2004.

[132] E. Laure, C. Gr, S. Fisher, A. Frohner, P. Kunszt, A. Krenek, O. Mulmo, F. Pacini, F. Prelz, J. White, M. Barroso, P. Buncic, R. Byrom, L. Cornwall, M. Craig, A. D. Meglio, A. Djaoui, F. Giacomini, J. Hahkala, F. Hemmer, S. Hicks, A. Edlund, A. Maraschini, R. Middleton, M. Sgaravatto, M. Steenbakkers, J. Walk, and A. Wilson. Programming the Grid with gLite. In *Computational Methods in Science and Technology*, page 2006, 2006.

[133] C. Lee, S. Matsuoka, D. Talia, A. Sussman, M. Mueller, G. Allen, and J. Saltz. A Grid Programming Primer. Technical report, Global Grid Forum Programming Model Working Group, Aug. 2001.

[134] S. Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[135] D. S. Linthicum. *Cloud Computing and SOA Convergence in Your Enterprise: A Step-by-Step Guide*. Addison-Wesley Professional, 1st edition, 2009.

[136] F. Lordan, E. Tejedor, J. Ejarque, R. Rafanell, J. Álvarez, F. Marozzo, D. Lezzi, R. Sirvent, D. Talia, and R. M. Badia. ServiceSs: an interoperable programming framework for the Cloud. *Springer Journal of Grid Computing, Special Issue on Interoperability, Federation, Frameworks and Application Programming Interfaces for IaaS Clouds*.

[137] D. A. Mallón, G. L. Taboada, J. Touriño, and R. Doallo. NPB-MPJ: NAS Parallel Benchmarks Implementation for Message-Passing in Java. In *Proc. 17th Euromicro Intl. Conf. on Parallel, Distributed, and Network-Based Processing (PDP'09)*, pages 181–190, Weimar, Germany, 2009.

[138] V. Marjanovic, J. Labarta, E. Ayguadé, and M. Valero. Effective communication and computation overlap with hybrid MPI/SMPSs. In *Proceedings of the 15th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 337–338, Bangalore, India, 2010.

[139] M. D. McIlroy. *Mass Produced Software Components*, pages 138–155. NATO Scientific Affairs Division: Brussels, 1969.

[140] P. E. McKenney. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* kernel.org, Corvallis, OR, USA, 2012. Available: `http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html`.

[141] P. Mell and T. Grance. The NIST Definition of Cloud Computing. Recommendations of the National Institute of Standards and Technology. *Nist Special Publication*, 145(6):1–2, 2011.

[142] P. Missier, S. Soiland-Reyes, S. Owen, W. Tan, A. Nenadic, I. Dunlop, A. Williams, T. Oinn, and C. Goble. Taverna, reloaded. In M. Gertz, T. Hey, and B. Ludaescher, editors, *SSDBM 2010*, Heidelberg, Germany, June 2010.

[143] F. Montesi, C. Guidi, and G. Zavattaro. Composing Services with JOLIE. In *Proceedings of the Fifth European Conference on Web Services*, pages 13–22, Washington, DC, USA, 2007. IEEE Computer Society.

[144] MPI Forum. MPI-Forum: A Message Passing Interface Standard. `http://www.mpi-forum.org`.

[145] F. Nadeem, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-l. Truong, and A. Villazon. ASKALON: A Development and Grid Workflows. *Workflows for eScience*, page 450–471, 2007.

[146] R. W. Numrich and J. Reid. Co-Array Fortran for parallel programming. *ACM Fortran Forum*, 17(2):1 – 31, 1998.

[147] W. O'Mullane, X. Luri, P. Parsons, U. Lammers, J. Hoar, and J. Hernandez. Using Java for distributed computing in the Gaia satellite data processing, European Space Agency Gaia mission. *CoRR*, abs/1108.0355, 2011.

[148] OpenMP Specifications. Openmp application programing interface. v3.0. `http://www.openmp.org/mp-documents/spec30.pdf`, May 2008.

[149] N. Parlavantzas, M. Morel, V. Getov, F. Baude, and D. Caromel. Performance and Scalability of a Component-Based Grid Application. In *9th Int. Workshop on Java for Parallel and Distributed Computing, in conjunction with the IEEE IPDPS conference*, April 2007.

[150] C. Pedrinaci and J. Domingue. Toward the Next Wave of Services: Linked Services for the Web of Data. *J. UCS*, 16(13):1694–1719, 2010.

[151] J. M. Perez, R. M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing*, pages 142–151, 2008.

[152] J. M. Perez, P. Bellens, R. M. Badia, and J. Labarta. CellSs: Making it easier to program the cell broadband engine processor. *IBM Journal of Research and Development*, 51(5), August 2007.

[153] G. F. Pfister. *In Search of Clusters (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[154] M. Philippsen. A survey of concurrent object-oriented languages. *Concurrency - Practice and Experience*, 12(10):917–980, 2000.

[155] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical task based programming with StarSs. *International Journal of High Performance Computing*, 23(3):284–299, August 2009.

[156] R. Rafanell. Extensió de COMP Superscalar. Projecte de Fi de Carrera, Universitat Autònoma de Barcelona.

[157] G. Raines. Cloud Computing and SOA, Service-Oriented Architecture Series. Technical report, The MITRE Corporation, October 2009.

[158] Java Remote Method Invocation. `http://www.oracle.com`.

[159] R. Royo, J. López, D. Torrents, and J. Gelpi. A BioMoby-based workflow for gene detection using sequence homology. In *International Supercomputing Conference (ISC'08), Dresden (Germany)*, 2008.

[160] W. Schulte and N. Tillmann. Automatic parallelization of programming languages: past, present and future. In *Proceedings of the 3rd International Workshop on Multicore Software Engineering*, IWMSE '10, pages 1–1, New York, NY, USA, 2010. ACM.

[161] D. B. Skillicorn and D. Talia. Models and Languages for Parallel Computation. *ACM Computing Surveys*, 30(2):123–169, 1998.

[162] E. Stewart. High Performance Java for Compute Intensive Applications, Visual Numerics Java Trends. *Java Developer's Journal*, Dec. 2007.

[163] A. Streit, D. Erwin, T. Lippert, D. Mallmann, R. Menday, M. Rambadt, M. Riedel, M. Romberg, B. Schuller, and P. Wieder. *Unicore — From project results to production grids*, volume 14 of *Advances in Parallel Computing*, pages 357–376. Elsevier, 2005.

[164] Supercomputing Technologies Group. MIT Laboratory for Computer Science. *Cilk 5.4.6 Reference Manual*, April 2005. `http://supertech.lcs.mit.edu/cilk`.

[165] G. L. Taboada, S. Ramos, R. R. Exposito, J. Touriño, and R. Doallo. Java in the High Performance Computing arena: Research, practice and experience. *Science of Computer Programming*, 2011 (In press http://dx.doi.org/10.1016/j.scico.2011.06.002).

[166] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka. Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing. *Journal of Grid Computing*, 1(1):41–51, 2003.

[167] I. Taylor, M. Shields, I. Wang, and A. Harrison. Visual Grid Workflow in Triana. *Journal of Grid Computing*, 3(3-4):153–169, September 2005.

[168] E. Tejedor and R. M. Badia. COMP Superscalar: Bringing GRID Superscalar and GCM Together. In *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, CCGrid '08, Lyon, France, pages 185–193, May 2008.

[169] E. Tejedor, R. M. Badia, R. Royo, and J. L. Gelpí. Enabling HMMER for the Grid with COMP Superscalar. In *Proceedings of the 10th International Conference on Computational Science 2010*, ICCS '10, Amsterdam, The Netherlands, May 2010.

[170] E. Tejedor, J. Ejarque, F. Lordan, R. Rafanell, J. Álvarez, D. Lezzi, R. Sirvent, and R. M. Badia. A Cloud-unaware Programming Model for Easy Development of Composite Services. In *Proceedings of the 3rd IEEE International Conference on Cloud Computing Technology and Science*, CloudCom '11, Athens, Greece, November 2011.

[171] E. Tejedor, M. Farreras, D. Grove, R. M. Badia, G. Almasi, and J. Labarta. ClusterSs: a Task-based Programming Model for Clusters. In *Proceedings of the 20th International ACM Symposium on High Performance Distributed Computing*, HPDC '11, San Jose, California, USA, pages 267–268, June 2011.

[172] E. Tejedor, M. Farreras, D. Grove, R. M. Badia, G. Almasi, and J. Labarta. A high-productivity task-based programming model for clusters. *Concurrency and Computation: Practice and Experience*, 24(18):2421–2448, 2012.

[173] E. Tejedor, F. Lordan, and R. M. Badia. Exploiting Inherent Task-Based Parallelism in Object-Oriented Programming. In *Proceedings of the 12th IEEE/ACM International Conference on Grid Computing*, GRID '11, Lyon, France, pages 74–81, September 2011.

[174] D. Thain, T. Tannenbaum, and M. Livny. Condor and the Grid. In F. Berman, G. Fox, and T. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., December 2002.

[175] R. V. van Nieuwpoort, G. Wrzesińska, C. J. Jacobs, and H. E. Bal. Satin: A high-level and efficient grid programming model. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(3):1–39, 2010.

[176] A. Vaqué, Í. Goiri, J. Guitart, and J. Torres. EMOTIVE Cloud: The BSC's IaaS Open Source Solution for Cloud Computing. In L. Vaquero, J. Cáceres, and J. Hierro, editors, *Open Source Cloud Computing Systems: Practices and Paradigms*, pages 44–60. IGI Global, 2012.

[177] C. Vecchiola, X. Chu, and R. Buyya. Aneka: A Software Platform for .NET-based Cloud Computing. *Computing Research Repository*, abs/0907.4, 2009.

[178] R. Virding, C. Wikström, and M. Williams. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1996.

[179] Volker Hoyer et al. The FAST Platform: An Open and Semantically-Enriched Platform for Designing Multi-channel and Enterprise-Class Gadgets. In *International Conference on Service Oriented Computing*, pages 316–330, 2009.

[180] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. In *19th International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, 1992.

[181] D. W. Walker, M. Li, O. F. Rana, M. S. Shields, and Y. Huang. The software architecture of a distributed problem-solving environment. *Concurrency: Practice and Experience*, 12(15):1455–1480, 2000.

[182] J. P. Walters. MPI-HMMER. `http://code.google.com/p/mpihmmer/`.

[183] J. P. Walters, R. Darole, and V. Chaudhary. Improving MPI-HMMER's scalability with parallel I/O. *Parallel and Distributed Processing Symposium, International*, 0:1–11, 2009.

[184] M. Wilde, I. Foster, K. Iskra, P. Beckman, Z. Zhang, A. Espinosa, M. Hategan, B. Clifford, and I. Raicu. Parallel scripting for applications at the petascale and beyond. *Computer*, 42:50–60, 2009.

[185] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. In ACM, editor, *ACM 1998 Workshop on Java for High-Performance Network Computing*, New York, NY 10036, USA, 1998. ACM Press.

Appendix A

Applications

A.1 Hmmpfam - Java StarSs

Main program of the Hmmpfam application for Java StarSs. Code shown in next page.

```
public static void main(String args[]) throws Exception {
    // Parameter parsing
    String dbName = args[0];
    String seqsName = args[1];
    String outputName = args[2];
    File fSeq = new File(seqsName);
    File fDB = new File(dbName);

    int numDBFrags = Integer.parseInt(args[3]);
    int numSeqFrags = Integer.parseInt(args[4]);
    List<String> dbFrags = new ArrayList<String>(numDBFrags);
    List<String> seqFrags = new ArrayList<String>(numSeqFrags);

    CommandLineArgs clArgs = new CommandLineArgs(args, 5);
    String finalArgs = clArgs.getArgs();


    /* FIRST PHASE
     * Segment the database file, the query sequences file or both
     */

    split(fDB, fSeq, dbFrags, seqFrags, numDBFrags, numSeqFrags);


    /* SECOND PHASE
     * Launch hmmpfam for each pair of seq - db fragments
     */

    int numReports = numDBFrags * numSeqFrags;
    String[] outputs = new String[numReports];
    int i = 0;
    for (String dbFrag : dbFrags) {
        for (String seqFrag : seqFrags) {
            outputs[i] = "report" + i + ".out";
            hmmpfam(finalArgs, dbFrag, seqFrag, outputs[i]);
            i++;
        }
    }


    /* THIRD PHASE
     * Merge all output in a single file
     */

    for (int gap = 1; gap < numReports; gap *= 2) {
        for (int pos = 0; pos < numReports; pos += 2 * gap)
            if (pos + gap < numReports)
                merge(outputs[pos], outputs[pos + gap]);
    }


    // Result is in file outputs[0]
    prepareResultFile(outputs[0], outputName, dbName, seqsName);
}
```

Figure A.1: Main program of the Hmmpfam application for Java StarSs.

A.2 Discrete - Java StarSs

Main program, task selection interface and example of a task graph generated by the Discrete application with Java StarSs.

A.2.1 Main Program

```
public static void main(String args[]) throws Exception {
    // Parameter parsing
    String binDir = args[0];
    String dataDir = args[1];
    String structDir = args[2];
    String tmpDir = args[3];
    String scoreDir = args[4];
    readParams(dataDir);

    // Generate coordinate and topology files for each structure
    for (int i = 1; i <= N; i++) {
        String pdbFile = structDir + "/1B6C_" + i + ".pdb";
        String recFile = tmpDir + "/receptor_" + i;
        String ligFile = tmpDir + "/ligand_" + i;
        String topFile = tmpDir + "/topology_" + i;
        String crdFile = tmpDir + "/coordinates_" + i;

        genReceptorLigand(pdbFile, binDir, recFile, ligFile);
        dmdSetup(recFile, ligFile, binDir, dataDir, topFile, crdFile);
    }

    String pydockFile = dataDir + PYDOCK;
    Queue<String> coeffList = new LinkedList<String>();
    Queue<String> list = new LinkedList<String>();

    // Parameter sweeping
    for (int i = 1; i <= STEPS; i++) {
        double fvdw = i * FVDW_STEP;
        for (int j = 1; j <= STEPS; j++) {
            double fsolv = j * FSOLV_STEP;
            for (int k = 1; k <= STEPS; k++) {
                double eps = k * EPS_STEP;
                String paramFile = genParamFile(fvdw, fsolv, eps);

                // N simulations, one for each structure
                for (int ii = 1; ii <= N; ii++) {
                    String topFile = tmpDir + "/topology_" + ii;
                    String crdFile = tmpDir + "/coordinates_" + ii;
                    String averageFile = tmpDir + "/average_" + UUID.randomUUID();
                    list.add(averageFile);
                    simulate(paramFile, topFile, crdFile, natom, binDir, dataDir, averageFile);
                }
```

```
                // Merge all averages in a single file
                while (list.size() > 1) {
                    Queue<String> listAux = new LinkedList<String>();
                    while (list.size() > 1) {
                        String a1 = list.poll();
                        String a2 = list.poll();
                        merge(a1, a2);
                        listAux.add(a1);
                    }
                    if (list.size() == 1) listAux.add(list.peek());
                    list = listAux;
                }

                String scoreFile = scoreDir + "/score_" + fvdw + "_"
                                   + fsolv + "_" + eps + ".score";
                String coeffFile = tmpDir + "/coeff_" + UUID.randomUUID();
                coeffList.add(coeffFile);

                // Generate the score file and calculate the final coefficient
                evaluate(list.poll(), pydockFile, fvdw, fsolv, eps, scoreFile, coeffFile);
            }
        }
    }

    // Find the min coefficient of all configurations
    while (coeffList.size() > 1) {
        String c1 = coeffList.poll();
        String c2 = coeffList.poll();
        min(c1, c2);
        coeffList.add(c1);
    }
}
```

Figure A.2: Main program of the Discrete application for Java StarSs.

A.2.2 Task Selection Interface

```
public interface DiscreteItf {
    @Method(declaringClass = "worker.discrete.DiscreteImpl")
    void genReceptorLigand(
        @Parameter(type = FILE) String pdbFile,
        String binDir,
        @Parameter(type = FILE, direction = OUT) String recFile,
        @Parameter(type = FILE, direction = OUT) String ligFile
    );

    @Method(declaringClass = "worker.discrete.DiscreteImpl")
    void dmdSetup(
        @Parameter(type = FILE) String recFile,
        @Parameter(type = FILE) String ligFile,
        String binDir,
        String dataDir,
        @Parameter(type = FILE, direction = OUT) String topFile,
        @Parameter(type = FILE, direction = OUT) String crdFile
    );

    @Method(declaringClass = "worker.discrete.DiscreteImpl")
    void simulate(
        @Parameter(type = FILE) String paramFile,
        @Parameter(type = FILE) String topFile,
        @Parameter(type = FILE) String crdFile,
        String natom,
        String binDir,
        String dataDir,
        @Parameter(type = FILE, direction = OUT) String average
    );

    @Method(declaringClass = "worker.discrete.DiscreteImpl")
    void merge(
        @Parameter(type = FILE, direction = INOUT) String f1,
        @Parameter(type = FILE) String f2
    );

    @Method(declaringClass = "worker.discrete.DiscreteImpl")
    void evaluate(
        @Parameter(type = FILE) String averageFile,
        @Parameter(type = FILE) String pydockFile,
        double fvdw,
        double fsolv,
        double eps,
        @Parameter(type = FILE, direction = OUT) String scoreFile,
        @Parameter(type = FILE, direction = OUT) String coeffFile
    );

    @Method(declaringClass = "worker.discrete.DiscreteImpl")
    void min(
        @Parameter(type = FILE, direction = INOUT) String f1,
        @Parameter(type = FILE) String f2
    );
}
```

Figure A.3: Task selection interface of the Discrete application for Java StarSs.

A.2.3 Task Graph

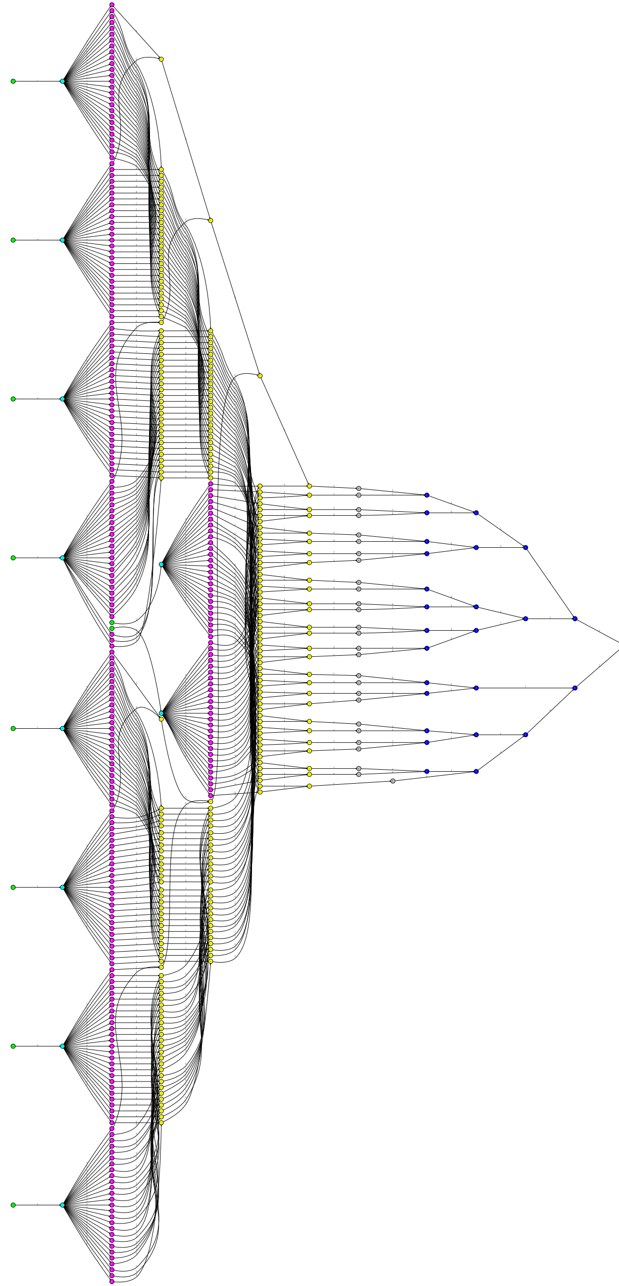


Figure A.4: Graph generated by Java StarSs for Discrete; input parameters: 10 structures, 27 different configurations of EPS, FSOLV and FVDW.

A.3 Gene Detection - Java StarSs

Main program and task selection interface of the Gene Detection composite service developed with Java StarSs.

A.3.1 Main Program

```
public class GeneDetection {

    private static final String NAMESPACE = "ENSEMBL";
    private static final int NALIGN = 100;

    @Orchestration
    public String detectGenes(String genome, String sequence) {

        /* ##### Genome DB formatting ##### */
        String genomeNCBI = genome + "_NCBI.zip";
        runNCBIFormatdb(genome, genomeNCBI);
        String genomeCNA = genome + "_CNA";
        CommentedNASequence cnaProperties
            = fromFastaToCommentedNASequence(genome, genomeCNA);

        /* ##### Sequences retrieval ##### */
        FASTA fastaSeq = loadSequenceFromFile(sequence);
        RunNCBIBlastpParameters params = setupNCBIBlastpParams(NALIGN));
        BLASTText report = runNCBIBlastp(fastaSeq, params);
        BlastIDs bIds = parseBlastIDs(report);
        List<NemusObject> seqIds = bIds.getIds(); // Synchronisation
        int numSeqs = seqIds.size();
        FASTA[] fastaSeqs = new FASTA[numSeqs];
        int i = 0;
        for (NemusObject seqId : seqIds) {
            BioTools btService = new BioTools();
            btService.loadAminoAcidSequence(seqId);
            fastaSeqs[i++] = btService.fromGenericSequenceToFasta(seqId);
        }

        /* ##### Gene search ##### */
        BLASTText[] blastResults = new BLASTText[numSeqs];
        for (i = 0; i < numSeqs; i++) {
            blastResults[i] = runNCBIBlastAgainstDBFromFASTA(
                genomeNCBI, fastaSeqs[i], setupNCBIBlastParameters());
        }
        for (int next = 1; next < numSeqs; next *= 2) {
            for (int result = 0; result < numSeqs; result += 2 * next) {
                if (result + next < numSeqs) {
                    blastResults[result].mergeBlastResults(blastResults[result+next]);
                }
            }
        }
        Bl2GAnnotations bl2gAnnots = runBlast2Gene(
            blastResults[0], setupBlast2GeneParameters(), Database.UNIREF_90);
        Bl2GAnnotations overlapAnnots = overlappingFromBL2G(bl2gAnnots);
```

```
        /* ##### GeneWise ##### */
        List<BL2GAnnotation> notOverlappedRegions =
            overlapAnnots.getAnnots(); // Synchronisation
        int numRegions = notOverlappedRegions.size();
        GenewiseReport[] gwResults = new GenewiseReport[numRegions];
        for (BL2GAnnotation reg : notOverlappedRegions) {
            FASTA seq = getSequence(reg.getProtID().getValue(), fastaSeqs);
            gwResults[i++] = runGenewise(genomeCNA, cnaProperties, reg, seq);
        }
        for (int next = 1; next < numRegions; next *= 2) {
            for (int result = 0; result < numRegions; result += 2 * next) {
                if (result + next < numRegions) {
                    gwResults[result].mergeGenewiseResults(gwResults[result+next]);
                }
            }
        }

        String gwReport = gwResults[0].getGff().getValue(); // Synchronisation
        return gwReport;
    }

    ...
}
```

Figure A.5: Main program of the Gene Detection composite for Java StarSs.

A.3.2 Task Selection Interface

```
public interface GeneDetectionItf {
    /* METHODS */

    @Method(declaringClass = "core.genedetect.GeneDetectMethods")
    void runNCBIFormatdb(
        String genomeName,
        @Parameter(type = FILE, direction = OUT) String genomeFile
    );

    @Method(declaringClass = "core.genedetect.GeneDetectMethods")
    CommentedNASequence fromFastaToCommentedNASequence(
        String genomeName,
        @Parameter(type = FILE, direction = OUT) String genomeFile
    );

    @Method(declaringClass = "core.genedetect.GeneDetectMethods")
    BLASTText runNCBIBlastp(
        FASTA fastaSeq,
        RunNCBIBlastpParameters params
    );

    @Method(declaringClass = "core.genedetect.GeneDetectMethods")
    BLASTText runNCBIBlastAgainstDBFromFASTA(
        @Parameter(type = FILE) String blastDBFile,
        FASTA fasta,
        RunNCBIBlastAgainstDBFromFASTAParameters params
    );
```

```
    @Method(declaringClass = "core.genedetect.BLASTText")
    void mergeBlastResults(
        BLASTText report
    );

    @Method(declaringClass = "core.genedetect.GeneDetectMethods")
    GenewiseReport runGenewise(
        @Parameter(type = FILE) String genomeCNAFile,
        CommentedNASequence cnaProperties,
        BL2GAnnotation region,
        FASTA sequence
    );

    @Method(declaringClass = "core.genedetect.GenewiseReport")
    void mergeGenewiseResults(
        GenewiseReport report
    );

    /* SERVICES */

    @Service(namespace = "http://genedetect.core", name = "BioTools",
             port = "BioToolsPort")
    BlastIDs parseBlastIDs(
        BLASTText report
    );

    @Service(namespace = "http://genedetect.core", name = "BioTools",
             port = "BioToolsPort")
    void loadAminoAcidSequence(
        NemusObject seqId
    );

    @Service(namespace = "http://genedetect.core", name = "BioTools",
             port = "BioToolsPort")
    FASTA fromGenericSequenceToFasta(
        NemusObject seqId
    );

    @Service(namespace = "http://genedetect.core", name = "BioTools",
             port = "BioToolsPort")
    Bl2GAnnotations runBlast2Gene(
        BLASTText blastResult,
        RunBlast2GeneParameters params,
        Database db
    );

    @Service(namespace = "http://genedetect.core", name = "BioTools",
             port = "BioToolsPort")
    Bl2GAnnotations overlappingFromBL2G(
        Bl2GAnnotations annots
    );

}
```

Figure A.6: Task selection interface of the Gene Detection composite for Java StarSs.

Appendix B

Resource Description

For the Java StarSs runtime to know the Grid/Cluster/Cloud resources at its disposal, it needs to be provided with a couple of XML configuration files that describe those resources. The next subsections present such configuration files.

B.1 Resources File

The resources file specifies a set of resources and their capabilities. Figure B.1 illustrates the definition of four different kinds of resources:

- *Physical machine*: the description of a physical machine includes its hardware and software capabilities, such as processor details, operating system, memory and storage sizes and software installed. These capabilities are used by the runtime to match the task constraints, if any, defined in the task selection interface (see an example in Figure 4.11 of Chapter 4). The XML tags are based on the Information Modeling standard proposal [41] by the Open Grid Services Architecture group of the Open Grid Forum.

- *Grid front-end node*: in a Grid scenario, the runtime can interact with a front-end node that provides access to a set of resources. This type of resource definition can also specify capabilities, in case they are known for the worker resources behind the front-end. In the example, a GridFTP server provided by the Grid site is defined.

- *Service instance*: this designs an instance of a web service hosted by a given server. Its tags are used to match the annotation of a service task in the task selection interface (see the service tasks declared in Figure A.6).

- *Cloud provider*: when working in Cloud environments, this kind of resource provides details about a particular Cloud provider, namely the connector that implements the interaction with the provider, the images available and the instance types that can be requested. This information is used by the runtime to dynamically create VMs on the provider.

```
<ResourceList>

  <Resource Name=”s05c2b14-gigabit1”>
    <Capabilities>
      <Processor>
        <Architecture>PPC</Architecture>
        <Speed>2.3</Speed>
        <CPUCount>4</CPUCount>
      </Processor>
      <OS>
        <OSType>Linux</OSType>
      </OS>
      <StorageElement>
        <Size>36</Size>
      </StorageElement>
      <Memory>
        <PhysicalSize>8</PhysicalSize>
      </Memory>
      <ApplicationSoftware>
        <Software>GeneWise</Software>
        <Software>BLAST</Software>
      </ApplicationSoftware>
    </Capabilities>
  </Resource>

  <Resource Name=”brgw1.renci.org:2119/jobmanager-pbs”>
    <Capabilities>
      ...
      <StorageElement>
        <Server name=”brgw1.renci.org” dir=”/home/engage/compss/”/>
      </StorageElement>
      ...
    </Capabilities>
  </Resource>

  <Service wsdl=”http://bscgrid05.bsc.es:20390/biotools/biotools?wsdl”>
    <Name>BioTools</Name>
    <Namespace>http://genedetect.core</Namespace>
    <Port>BioToolsPort</Port>
  </Service>

  <CloudProvider name=”Amazon”>
    <Connector>integratedtoolkit.connectors.amazon.EC2</Connector>
    <ImageList>
      <Image name=”ami-7b85820f”>
        <OS>
          <OSType>Linux</OSType>
        </OS>
        <ApplicationSoftware>
          <Software>Discrete</Software>
        </ApplicationSoftware>
      </Image>
    </ImageList>
```

```
        <InstanceTypes>
            <Resource Name="t1.micro">
                <Capabilities>
                    <Processor>
                        <CPUCount>1</CPUCount>
                    </Processor>
                    <StorageElement>
                        <Size>30</Size>
                    </StorageElement>
                    <Memory>
                        <PhysicalSize>0.5</PhysicalSize>
                    </Memory>
                </Capabilities>
            </Resource>
            <Resource Name="m1.small"> ...
            <Resource Name="m1.medium"> ...
            <Resource Name="m1.large"> ...
            <Resource Name="m1.xlarge"> ...
        </InstanceTypes>
    </CloudProvider>

</ResourceList>
```

Figure B.1: Snippet of a resources file.

B.2 Project File

The project file contains the resources to be used in a particular execution of a Java StarSs application. The resources in this file must be a subset of those appearing in the resources file. Figure B.2 shows an example of a project file, where the four resources in Figure B.1 are selected.

The project file specifies some information related to the execution of the application. This includes the number of slots offered by the resource (LimitOfTasks), i.e. the number of concurrent tasks that can be run in that resource. For Cloud providers, a useful feature is the possibility to specify packages to deploy on a newly created VM before submitting tasks to it (Package).

```xml
<Project>

    <Worker Name="s05c2b14-gigabit1">
        <InstallDir>/home/bsc19121/IT_worker/</InstallDir>
        <WorkingDir>/home/bsc19121/IT_worker/files/</WorkingDir>
        <User>bsc19121</User>
        <LimitOfTasks>4</LimitOfTasks>
    </Worker>

    <Worker Name="brgw1.renci.org:2119/jobmanager-pbs">
        <InstallDir>/osg/osg-app/IT_worker/</InstallDir>
        <WorkingDir>/osg/osg-data/</WorkingDir>
        <LimitOfTasks>2</LimitOfTasks>
    </Worker>

    <Worker Name="http://bscgrid05.bsc.es:20390/biotools/biotools?wsdl">
        <LimitOfTasks>2</LimitOfTasks>
    </Worker>

    <Cloud>
        <Provider name="Amazon">
            <LimitOfVMs>4</LimitOfVMs>
            <Property>
                <Name>Placement</Name>
                <Value>eu-west-1a</Value>
            </Property>
            <Property>
                <Name>KeyPair name</Name>
                <Value>keypair_enric</Value>
            </Property>
            ...
            <ImageList>
                <Image name="ami-7b85820f">
                    <InstallDir>/aplic/COMPSs/</InstallDir>
                    <WorkingDir>/home/ec2-user/</WorkingDir>
                    <User>ec2-user</User>
                    <Package>
                        <Source>/home/etejedor/genedetect.tar.gz</Source>
                        <Target>/home/ec2-user</Target>
                    </Package>
                </Image>
            </ImageList>
            <InstanceTypes>
                <Resource name="m1.medium" />
            </InstanceTypes>
        </Provider>
    </Cloud>

</Project>
```

Figure B.2: Snippet of a project file.