Available online at www.sciencedirect.com
ScienceDirect
Future Generation Computer Systems 23 (2007) 587–605
FGCS FUTURE GENERATION COMPUTER SYSTEMS
www.elsevier.com/locate/fgcs

Specification-correct and scalable coordination of Grid applications

Radu Prodan*

Institute for Computer Science, University of Innsbruck, Technikerstraße 22, A-6020 Innsbruck, Austria

Received 28 February 2006; received in revised form 12 July 2006; accepted 16 September 2006
Available online 9 November 2006

Abstract

The so called "invisible Grid", transparent to application developers, is still far from being a reality. One reason is that the workflow model, which emerged as a widely accepted paradigm for high-level composition of Grid applications, is based on a low-level imperative programming model prone to programming errors. The issue of developing correct (bug-free) Grid applications has not been addressed by the community.

We propose a new model for building Grid applications based on two programming phases: (1) formal functional specification, written by the application developer not interested in any Grid-related issues; (2) imperative workflow-based coordination, written by the computer scientist, which ports and efficiently executes the specification onto the Grid. A correctness checker automatically connects both parts at compile-time and insures the correct execution of the workflow coordination with respect to the formal specification.

We validate our approach for three scientific applications and show real-world experimental results that demonstrate the scalability of our coordination model and the fact that the overhead introduced by our correctness checker is insignificant when compared to the latencies exhibited by the Grid middleware software.
© 2006 Elsevier B.V. All rights reserved.

Keywords: Specification techniques; Coherence and coordination; Workflow management; Functional constructs; Correctness proofs; Performance evaluation

1. Introduction

The workflow model has emerged as one of the most attractive paradigms for programming Grid applications. As a consequence, significant efforts are taking place currently in academia and industry to define appropriate languages that express workflows at a high-level of abstraction, thus completely shielding the Grid from the application programmer [7,12,14,19,9]. Despite these solid efforts, we believe that the current outcomes are still far from fulfilling the ultimate goal of scientists fof having an *invisible Grid*, which they can program with ease and which is error-free.

Typically, Grid workflows are defined as a collection of off-the-shelf activities or components, interconnected through control and data flow dependencies into a directed (sometimes acyclic) graph structure. The control flow dependencies express the execution order of activities, while the data dependencies express the communication between activities.

1.1. Error-prone programming

Similar to Fortran, C, or Java, workflow languages are based on the imperative model of computation. *Imperative programming* is traditionally used in scientific computing due to its *performance* advantage. Programs are modelled as a workflow of instructions that reuse *data stores* (e.g. registries, cache, main memory, hard disk, storage systems) through destructive assignments. A skilful reuse of such stores underlies the art of achieving high performance, but is at the same time the most important source of *programming errors*. A data store that contains a *wrong value* during the execution of a program is often called a *bug*.

Existing high-level Grid workflow languages are prone to programming errors at the same low-level of abstraction as imperative assembly languages. An erroneous or missing control flow dependency between two activities, a wrong data flow dependency, or a data port containing a wrong data package represents a bug.

Functional programming, based on the *expression evaluation model*, appears to be missing several constructs often

* Tel.: +43 512 507 6445; fax: +43 512 507 2758.
E-mail address: radu@dps.uibk.ac.at.

0167-739X/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.future.2006.09.008

considered essential to an imperative languages. For example, in strict functional programming, there is no explicit memory allocation and no explicit variable assignment. However, these operations occur automatically when a function is invoked. A memory allocation occurs to make space for the parameters and the return value. An assignment occurs to copy the parameters into this newly allocated space and to copy the return value back into the calling function. Both operations can only occur on function entry and exit, so any side effects of function evaluation are eliminated. By disallowing side effects in functions, the language provides referential transparency. This ensures that the result of a function will be the same for a given set of parameters no matter where, or when, it is evaluated. This is a big step for insuring the *program correctness*.

Moreover, functional languages allow for the deployment of techniques for verifying the program correctness, which are much more cumbersome when applied to programs written in imperative languages. *Inductive reasoning* remains, regardless of its inefficiency, a powerful technique to establish the correctness of a recursive function in two steps: (1) The base cases of the induction correspond to the *pre-* and *postconditions* that make no recursive calls; (2) The inductive step (the beauty) assumes that the recursive calls work correctly when showing that a case involving recursive calls is correct.

1.2. Low-level abstractions

Despite the declared goal of producing high-level workflow languages for building applications, we see the outcomes in the Grid community are very much similar to the traditional low-level imperative languages. Sequential programs, written in popular imperative languages like Fortran, C, or Java, are, in fact, *workflows of instructions*. The Grid renames the atomic unit of work from instruction to component or activity. The data ports used to communicate information between workflow activities are, in fact, a different names for data stores or *variables*. Data flow dependencies are, in fact, the results of traditional low-level compiler analysis results of *assignment statements*. In addition, to support the effective development of applications, Grid workflow languages introduce additional complex control flow constructs like `while`, `for`, `if`, and even `switch`. Workflow languages are therefore Turing-complete, and provide a rich type of system, which could be used to express any kind of algorithm (such as sorting or graph-based) at the same level of abstraction as traditional imperative languages like C, Fortran, or Java.

For example, a broad class of Grid applications are currently modelled as directed graph-based workflows (see Section 3) that contain loops. The representation of loops through `for` or `while` constructs is, however, the computer scientist's imperative view of applications, oriented towards high performance. Physicists, in contrast, naturally think and represent cyclic computations through store-free *recursive mathematical formulas*.

1.3. Semantics

The latest trend in the Grid research community advocates the use of semantic techniques (based on ontologies) to build Grid applications through automatic composition of off-the-shelf components. Moreover, the recent movement towards the Semantic Grid brings up a parallel between the Semantic Grid and the Semantic Web towards which researchers in natural language processing in particular, and in Artificial Intelligence (AI) in general, strive. Such a parallel brings up the issue of connecting the Grid community to the AI field, to start making use of tools and techniques that have already been shown to be useful for exploiting the Semantic Web. One of the most developed applications of the Semantic Web, which bears a striking resemblance to the ultimate goal of Grid developers, is the processing of available knowledge using inference engines. While different approaches may be applicable for the construction of inference engines (general logic based inference engines, higher order logic based, full first order logic based, description logic, or problem-solving methods), they all share a common feature: they are based on *logic*.

While logical languages gained popularity in Europe and Japan for various AI tasks, in the US the preference has traditionally been for functional programming languages, which offer just as powerful a tool for addressing tasks like semantic analysis and planning. A functional-based approach allows for an intuitive implementation based on inference rules, allowing a user to focus on *what* needs to be done, as opposed to *how* to do it (on the Grid), which is the case for imperative programming languages. We believe that this could help the transition towards an invisible Grid.

1.4. Outline

In this paper, we propose a new approach for programming the Grid based on the ideas of run-time checking [22], program verification, and result checking [3], and two-stage programming [15]. The model, summarised in Section 2 and illustrated throughout this paper in the context of a real-world material science application introduced in Section 3, is based on a full separation between a purely functional *formal specification* (see Section 4) and an *imperative workflow-based coordination* (see Section 5) of applications. A *correctness checker*, presented in Section 6, links the two parts at compile-time, and automatically insures that the low-level workflow coordination is executed correctly with respect to the mathematical specification. In Section 7, we show a concrete binding of the coordination model for AGWL [9], that is an Abstract Grid Workflow Language resembling many other modern XML-based efforts in the field. In Section 8, we introduce the ASKALON Grid application development and computing environment that represents the umbrella project within which we designed and implemented our approach. Experimental results discussed in Section 9 illustrate the scalability of my coordination language, and demonstrates the fact that the correctness checking overhead introduced by my model is negligible compared to the latencies of the Grid middleware software. Sections 10 and 11 present two additional application case studies for validating my approach. Section 12 compares my work against the most relevant related efforts in the field. Section 13 summarises the paper contributions and gives a brief outlook into future research.

Download English Version:

https://daneshyari.com/en/article/424998

Download Persian Version:

https://daneshyari.com/article/424998

Daneshyari.com