Computer Physics Communications 184 (2013) 456–468

Contents lists available at SciVerse ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

Multi-physics simulations using a hierarchical interchangeable software interface

Simon F. Portegies Zwart [a,*], Stephen L.W. McMillan [b], Arjen van Elteren [a], F. Inti Pelupessy [a], Nathan de Vries [a]

[a] Sterrewacht Leiden, P.O. Box 9513, 2300 RA Leiden, The Netherlands
[b] Department of Physics, Drexel University, Philadelphia, PA 19104, USA

ARTICLE INFO

Article history:
Received 11 June 2012
Received in revised form
17 September 2012
Accepted 20 September 2012
Available online 26 September 2012

Keywords:
Computer applications
Physical sciences and engineering
Astronomy
Computing methodologies: simulation modeling, and visualization
Distributed computing

ABSTRACT

We introduce a general-purpose framework for interconnecting scientific simulation programs using a homogeneous, unified interface. Our framework is intrinsically parallel, and conveniently separates all component numerical modules in memory. This strict separation allows automatic unit conversion, distributed execution of modules on different cores within a cluster or grid, and orderly recovery from errors. The framework can be efficiently implemented and incurs an acceptable overhead. In practice, we measure the time spent in the framework to be less than 1% of the wall-clock time. Due to the unified structure of the interface, incorporating multiple modules addressing the same physics in different ways is relatively straightforward. Different modules may be advanced serially or in parallel. Despite initial concerns, we have encountered relatively few problems with this strict separation between modules, and the results of our simulations are consistent with earlier results using more traditional monolithic approaches. This framework provides a platform to combine existing simulation codes or develop new physical solver codes within a rich "ecosystem" of interchangeable modules.

1. Introduction

Large-scale, high-resolution computer simulations dominate many areas of theoretical and computational science. The demand for such simulations has expanded steadily over the past decade, and is likely to continue to grow in coming years due to the increase in the volume, precision, and dynamic range of experimental data, as well as the widening spectral coverage of observations and laboratory experiments. Simulations are often used to mine and understand large observational and experimental datasets, and the quality of these simulations must keep pace with the increasingly high quality of experimental data.

In our own specialized discipline of computational astrophysics, numerical simulations have increased dramatically in both scope and scale over the past four decades. In the 1970s and 1980s, large-scale astrophysical simulations generally incorporated "mono-physics" solutions—in our case, the sub-disciplines of stellar evolution [1], gas dynamics [2], and gravitational dynamics [3]. A decade later, it became common to study phenomena combining a few different physics solvers [4]. Today's simulation environments incorporate multiple physical domains, and their nominal dynamic range often exceeds the standard numerical precision of available compilers and hardware [3].

Recent developments in hardware—in particular the rapidly increasing availability of multi-core architectures—have led to a surge in computer performance [5]. With the volume and quality of experimental data continuously improving, simulations expanding in scope and scale, and raw computational speed growing more rapidly than ever before, one might expect commensurate returns in the scientific results returned. However, a major bottleneck in modern computer modeling lies in the software, the growing complexity of which is evident in the increase in the number of code lines, the lengthening lists of input parameters, the number of underlying (and often undocumented) assumptions, and the expanding range of initial and boundary conditions.

Simulation environments have grown substantially in recent years by incorporating more detailed interactions among constituent systems, resulting in the need to incorporate very different physical solvers into the simulations, but the fundamental design of the underlying codes has remained largely unchanged since the introduction of object-oriented programming [6] and patterns [7]. As a result, maintaining and extending existing large-scale, multi-physics solvers has become a major undertaking. The legacy of design choices made long ago can hamper further development and expansion of a code, prevent scaling on large parallel computers, and render maintenance almost impossible. Even configuring, compiling, and running such a code has become a complex endeavor, not for the faint of heart. It has become increasingly difficult to reproduce simulation results, and independent code

* Corresponding author.
E-mail address: spz@strw.leidenuniv.nl (S.F. Portegies Zwart).

0010-4655/$ – see front matter
doi:10.1016/j.cpc.2012.09.024

verification and validation are rarely performed, even though all researchers agree that computer experiments require the same degree of reproducibility as is customary in laboratory settings.

We suggest that the root cause of much of this code complexity lies in the traditional approach to incorporating multi-physics components into a simulation—namely, solving the equations appropriate to all components in a single monolithic software suite, often written by a single researcher or research group. Such a solution may seem desirable from the standpoint of consistency and performance, but the resulting software generally suffers from all of the fundamental problems just described. In addition, integration of new components often requires sweeping redesign and redevelopment of the code. Writing a general multi-physics application from scratch is a major undertaking, and the stringent requirements of high-quality, fault-tolerant scientific simulation software render such code development by a single author almost impossible.

But why reinvent or re-engineer a monolithic suite of coupled mono-physics solvers when well-tested applications already exist to perform many or all of the necessary individual tasks? In many scientific communities there is a rich tradition of sharing scientific software. Many of these programs have been written by experts who have spent careers developing these codes and using them to conduct a wide range of numerical experiments. These packages are generally developed and maintained independently of one another. We refer to them collectively as "community" software. The term is intended to encompass both "legacy" codes that are still maintained but are no longer under active development, and new codes still under development to address physical problems of current interest. Together, community codes represent an invaluable, if incoherent, resource for computational science.

Coupling community codes raises new issues not found in monolithic applications. Aside from the wide range of physical processes, underlying equations, and numerical solvers they represent, these independent codes generally also employ a wide variety of units, input and output methods, file formats, numerical assumptions, and boundary conditions. Their originality and independence are strengths, but the lack of uniformity can also significantly reduce the "shelf life" of the software. In addition, directly coupling the very dissimilar algorithms and data representations used in different community codes can be a difficult task—almost as complex as rewriting the codes themselves. But whatever the internal workings of the codes, they are designed to represent a given domain of physics, and different codes may (and do in practice) implement alternate descriptions of the same physical processes. This suggests that integrating community codes should be possible using interfaces based on physical principles.

In this paper we present a comprehensive solution to many of the problems mentioned above, in the form of a software framework that combines remote function calls with physically based interfaces, and implements an object oriented data model, automatic conversion of units, and a state handling model for the component solvers, including error recovery mechanisms. Communication between the various solvers is realized via a centralized message passing framework, under the overall control of a high-level user interface. In Section 2, we name our framework MUSE, the MUlti-physics Software Environment. An example of the MUSE framework is presented in Section 3. In Section 4 we describe a production implementation of AMUSE, the astrophysics MUSE environment, which supports a wide range of programming languages and physical environments.

1.1. An historical perspective on MUSE

The basic concepts of MUSE are rooted in the earliest development of multi-scale software in computational astrophysics. The idea of combining codes within a flexible framework began with the NEMO project in 1986 at the Institute for Advanced Study (IAS) in Princeton [8,9]. NEMO was (and still is) aimed primarily at collisionless galactic dynamics. It used a uniform file structure to communicate data among its component programs.

The Starlab package [10,11], begun in 1993 (again at IAS), adopted the NEMO toolbox approach, but used pipes instead of files for communication among modules. The goal of the project was to combine dynamics and stellar and binary evolution for studies of collisional systems, such as star clusters. The stellar/binary evolution code SeBa [12] was combined with a high-performance gravitational stellar dynamics simulator. Because of the heterogeneous nature of the data, not all tools were aware of all data types (for example, the stellar evolution tools generally had no inherent knowledge of large-scale gravitational dynamics). As a result, the package used an XML-like tagged data format to ensure that no information was lost in the production pipeline—unknown particle data were simply passed unchanged from input to output.

The intellectual parent of (A)MUSE is the MODEST initiative, begun in 2002 at a workshop at the American Museum of Natural History in New York. The goal of that workshop was to formalize some of the ideas of modular software frameworks then circulating in the community into a coherent system for simulating dense stellar systems. Originally, MODEST stood for MOdeling DEnse STellar systems (star clusters and galactic nuclei). The name was later expanded, at the suggestion of Giampolo Piotto (Padova) to Modeling and Observing DEnse STellar systems. The MODEST web page can be found at http://www.manybody.org/modest. Since then, MODEST has gone on to provide a lively and long-lived forum for discussion of many topics in astrophysics. (A)MUSE is in many ways the software component of the MODEST community. An early example of MUSE-like code can be found in the proceedings of the MODEST-1 meeting [13].

Subsequent MODEST meetings discussed many new ideas for modular multiphysics applications [14, e.g.]. The basic MUSE architecture, as described in this paper, was conceived during the 2005 MODEST-6a workshop in Lund [15, Sweden]. The MUSE name, and the first lines of MUSE code, were created during MODEST-6e in Amsterdam in 2006, and expanded upon over the next 1–2 years. The "Noah's Ark" milestone (meeting our initial goal of having two independent modules for solving each particular type of physics) was realized in 2007, during MODEST-7f in Amsterdam and MODEST-7a in Split [16, Croatia]. The AMUSE project, short for for Astrophysics MUlti-purpose Software Environment, a re-engineered version of MUSE—"MUSE 2.0", building on the lessons learned during the previous 3 years—began at Leiden Observatory in 2009.

2. The MUSE framework

Each of the problems discussed in Section 1 could in principle be addressed by designing an entirely new suite of programs from scratch. However, this idealized approach fails to capitalize on the existing strengths of the field by ignoring the wealth of highly capable scientific software that has been developed over the last four or five decades. We will argue that it is more practical, and considerably easier, to introduce a generalized interface that connects existing solvers to a homogeneous and easily extensible framework.

At first sight, the approach of assimilating existing software into a larger framework would appear to be a difficult undertaking, particularly since community software may be written in a wide variety of languages, such as FORTRAN, C, and C++, and exhibits an enormous diversity of internal units and data structures. On the other hand, such a framework, if properly designed, could be relatively easy to use, since learning one simulation package