New Astronomy 34 (2015) 98–107

Contents lists available at ScienceDirect

New Astronomy

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

Mechanic: The MPI/HDF code framework for dynamical astronomy

Mariusz Słonina *, Krzysztof Goździewski, Cezary Migaszewski

Toruń Centre for Astronomy, Nicolaus Copernicus University, Gagarin Str. 11, 87-100 Torun, Poland

HIGHLIGHTS

- A new open-source MPI-based general-purpose parallel code framework is introduced.
- It reduces the software development, task management and data post-processing effort.
- It allows to parallelize serial software on a CPU-cluster.
- The provided API allows to use Monte Carlo and evolutionary algorithms.
- No parallel programming knowledge is required to use the framework.

ARTICLE INFO

Article history:
Received 25 October 2013
Received in revised form 27 January 2014
Accepted 25 May 2014
Available online 5 June 2014

Communicated by J. Makino

Keywords:
Numerical methods
Task management
Message Passing Interface
Hierarchical Data Format

ABSTRACT

We introduce the `Mechanic`, a new open-source code framework. It is designed to reduce the development effort of scientific applications by providing unified API (Application Programming Interface) for configuration, data storage and task management. The communication layer is based on the well-established Message Passing Interface (MPI) standard, which is widely used on variety of parallel computers and CPU-clusters. The data storage is performed within the Hierarchical Data Format (HDF5). The design of the code follows *core–module* approach which allows to reduce the user's codebase and makes it portable for single- and multi-CPU environments. The framework may be used in a local user's environment, without administrative access to the cluster, under the PBS or Slurm job schedulers. It may become a helper tool for a wide range of astronomical applications, particularly focused on processing large data sets, such as dynamical studies of long-term orbital evolution of planetary systems with Monte Carlo methods, dynamical maps or evolutionary algorithms. It has been already applied in numerical experiments conducted for Kepler-11 (Migaszewski et al., 2012) and *υ*Octantis planetary systems (Goździewski et al., 2013). In this paper we describe the basics of the framework, including code listings for the implementation of a sample user's module. The code is illustrated on a model Hamiltonian introduced by (Froeschlé et al., 2000) presenting the Arnold diffusion. The Arnold web is shown with the help of the MEGNO (Mean Exponential Growth of Nearby Orbits) fast indicator (Goździewski et al., 2008a) applied onto symplectic $SABA_n$ integrators family (Laskar and Robutel, 2001).

1. Introduction

In the field of dynamical astronomy several numerical techniques have been proposed to determine the nature of the phase space of planetary systems. The Monte Carlo methods (e.g., Holman and Wiegert, 1999), evolutionary algorithms (e.g., Goździewski et al., 2008b; Goździewski and Migaszewski, 2009) or dynamical maps (e.g., Froeschlé et al., 2000; Guzzo, 2005; Migaszewski et al., 2012; Goździewski et al., 2013) have become standard research tools for determining possible or permitted configurations, mass ranges or other physical data. These experiments usually require intensive tests of sets of initial conditions, that represent different orbital configurations. They involve direct numerical integrations of equations of motion to study long-term orbital evolution. To characterize the dynamical stability of orbital models, so called *fast chaos indicators* are often used (e.g., Goździewski et al., 2008a). These numerical tools make it possible to resolve efficiently whether a given solution is stable (quasi-periodic, regular) or unstable (chaotic) by following relatively short parts of the orbits. The fast indicators, like the Fast Lyapunov Indicator (FLI, Froeschlé et al., 2000), the Frequency Map Analysis (FMA, Laskar, 1993; Sidlichovský and Nesvorný, 1996), the Mean Exponential Growth factor of Nearby Orbits (MEGNO, Cincotta and Simó, 2000; Cincotta et al., 2003; Mestre et al., 2011), the

* Corresponding author. Tel./fax: +48 696023674.
E-mail addresses: m.slonina@astri.umk.pl (M. Słonina), k.gozdziewski@astri.umk.pl (K. Goździewski), c.migaszewski@astri.umk.pl (C. Migaszewski).

http://dx.doi.org/10.1016/j.newast.2014.05.006
1384-1076/

Spectral Number (SN, Michtchenko and Ferraz-Mello, 2001), are well known in the theory of dynamical systems (Barrio et al., 2009). In the past decade, they were intensively adapted to the planetary dynamics (e.g., Froeschlé et al., 1997; Robutel and Laskar, 2001; Goździewski et al., 2008a).

Depending on the dynamical model of a planetary system, its numerical setup and the chaos indicator used to represent the dynamical state, the simulation of a set of initial conditions may require large CPU resources. However, since each test may be understood as a separated numerical *task*, parallelization techniques may be used, with tasks distributed among available CPUs and evaluated in *parallel*. The basic approach relies on the *task farm* model, in which independent tasks are processed on *worker* nodes with the result collected by the *master*. From the technical point of view, this algorithm requires implementation of the CPU-communication layer, and should allow input preparation and result assembly for the post-processing. These issues are addressed in general-purpose distributed task management systems, like `HTCondor` (Fields, 1993), or `Workqueue` (Yu et al., 2010). Within such frameworks, the user-supplied, standalone executable code performing computations (*application*) is distributed over a computing pool. The input and output for each software instance is achieved via batch scripts (e.g. the `Makeflow` extension for the `Workqueue` package), making this approach application- and problem-dependent. This might be insufficient for large and long-term numerical tests, such as studying the dynamics of planetary systems. In particular, our recent work on Kepler-11 (Migaszewski et al., 2012) and νOctantis (Goździewski et al., 2013) systems required developing a new code framework, the `Mechanic`, dedicated to conducting massive parallel simulations. It has been turned out into general-purpose master–worker framework, built on the foundation of the Message Passing Interface (Pacheco, 1996). The `Mechanic` separates the numerical part of the user's code (*a module*) from its configuration, communication and storage layers (*a core*). This partition is achieved through the provided Application Programming Interface (API). On the contrary to `HTCondor` and `Workqueue` packages, the task preparation and result data storage is handled by the core of the framework. The final result is assembled into one datafile, which reduces the cost of post-processing large simulations. The storage layer is built on top of the universal HDF5 data format (The HDF5 Group, 2012). No MPI nor HDF5 programming knowledge is required to use the framework, which makes it possible to parallelize "scalar" codes relatively easily. The `Mechanic` may be used both system-wide as well as in a local user's environment under the control of job schedulers, such as PBS or Slurm.

This paper is structured as follows. We give a short overview of the `Mechanic` in Section 2. To explain programming concepts behind the framework we illustrate it with the help of the Hamiltonian model introduced by Froeschlé et al. (2000). It reveals the so called *Arnold web*, which represents a set of resonances of a quasi-integrable dynamical systems. It has been intensively studied in recent years (Cincotta, 2002; Lega et al., 2003; Guzzo et al., 2004; Froeschlé et al., 2005; Froeschlé et al., 2006), and applied to study long-term evolution of the outer Solar System (Guzzo, 2005; Guzzo, 2006). In Section 3 we give short theoretical background on this topic. The very fine details of the phase space obtained with the dedicated module for the `Mechanic` are presented in the Section 4. The technical implementation of the module is given in the Appendix A.

2. Overview of the framework

The `Mechanic` provides a skeleton code for common technical operations, including run-time configuration, memory and file management, as well as CPU communication. It has been developed to mimic the user's application flow in a problem-independent way (Listing 1). This is achieved via provided API, which allows to reduce the user's code to a *module* form, containing only its numerical part along with setup and storage specifications required to run it (Listing 2). The *core* of the framework loads the module dynamically during the runtime, performs the setup and storage stages according to these specs and executes the numerical part.

The benefit of this *core–module* approach comes both in data and task management. For instance, let us recall the concept of dynamical maps. The phase space of the dynamical system is mapped onto two-dimensional plane. Each point on that plane represents the dynamical state of the specific initial condition. From the technical point of view, computing the dynamical map requires execution of several numerical tasks that differs with the input, and assembling the result in an accessible way for post-processing. Assuming that each task (initial condition) is computed by a single instance of the application, the simulation requires preparing the input and collecting the result with the help of batch scripts. Although the `HTCondor` and `Workqueue` frameworks provide powerful task management tools, input and output data management is left to the user. The `Mechanic` framework works more like `Makeflow` (a `Workqueue` make engine), however, the user's code connected to the core is treated as a whole application with single output datafile and the input that may be prepared programatically according to the information associated with the current task.

The base master–worker algorithm with single input and output may be easily implemented with the minimum knowledge on the MPI programming. However, the purpose of the `Mechanic` is to reduce this development effort. With the help of the API, the user's code is separated from the task management layer. It allows to use different communication patterns between nodes in a computing pool (cluster) without modification of the module. In addition to the master–worker pattern, the master-only mode without MPI communication is provided, which behaves similar to a single-CPU application. Moreover, the API allows to implement different communication patterns, if required by the user's code.

The task assignment is performed within multidimensional grid and is governed through the API. Although this suits best the concept of dynamical maps, the API has been designed to support different assignment patterns, such as Monte Carlo methods. The key design concept of the `Mechanic` is a *task pool*. It represents a set of numerical tasks to perform for a particular setup (i.e. single dynamical map). The framework allows to create task pools dynamically depending on the results, with different configuration, storage and number of tasks. This helps to implement evolutionary algorithms and processing pipelines despite of number of CPUs involved in the simulation.

The result data, among with the run-time configuration, is assembled into one master datafile. Each task may hold unlimited number of multidimensional arrays of all native datatypes. Depending on the application requirements, the results obtained from all tasks may be combined in different modes. This includes *texture*, represented by a single dataset that follows the grid pattern (suitable for image-like results, such as dynamical maps), *group* (datasets are combined into separate groups per task), *list* (a spreadsheet-like dataset) and *pm3d* (dataset prepared for `pm3d` mode of Gnuplot). The usage of the single master file helps to reduce the post-processing effort. Numerous HDF-oriented applications are available (such as `h5py`), making it possible to use computation results independently of the host software.

For long-term simulations the checkpoint feature is important. Indeed, during the computations, the `Mechanic` provides the incremental snapshot file with the current state of the simulation. The file is self-contained, and includes run-time configuration, so