Parallel processing and visualization for results of molecular simulation problems

D. V. Puzyrkov <dpuzyrkov@gmail.com>
V. O. Podryga <pvictoria@list.ru>
S. V. Polyakov <polyakov@imamod.ru>
Keldysh Institute of Applied Mathematics (Russian Academy of Sciences)
Miusskaya sq., 4, Moscow, 125047, Russia

Abstract. In this paper authors presents “mmdlab” library for the interpreted programming language Python. This library allows to carry out reading, processing and visualization of the results of numerical calculations in the tasks of molecular simulation. Considering the large volume of data obtained from such simulations, there is a need in parallel realization of algorithms for processing those volumes. Parallel processing should be performed on multicore systems, such as common scientific workstation, and on super-computer systems and clusters, where the MD simulations were held. During the development process we have study the effectiveness of the Python language for such tasks, and we have examined the tools for it’s acceleration. As well, we studied multiprocessing capabilities and tools for cluster computation using this language. Also we have investigated the problems of receiving and processing the data, located on multiple computational nodes. This was prompted by the need to process the data, produced by parallel algorithm, that was executed on multiple computational nodes, and saves its output on each of them. As a tool for scientific visualization was chosen an open-source “Mayavi2” package. The developed ”mmdlab” library was used in the analysis of the results of MD simulation of the gas and metal plate interaction. As a result, we managed to observe the effect of adsorption in details, which is important for many practical applications.

Keywords: parallel processing; visualization; molecular dynamics; Python; Mayavi2

DOI: 10.15514/ISPRAS-2016-28(2)-15

Для цитирования: Puzyrkov D.V., Podryga V.O., Polyakov S.V. Parallel processing and visualization for results of molecular simulation problems. *Trudy ISP RAN/Proc. ISP RAS*, vol. 28, issue 2, 2016, pp. 221-242. DOI: 10.15514/ISPRAS-2016-28(2)-15

1. Introduction

Advances in computer technology and the rapid growth of computational capabilities significantly increased the possibilities of computational experiment (CE). In particular, nowadays it is already possible to study the properties and processes in complex systems on molecular and atomic levels, for example, using molecular dynamics (MD) approach. Mathematical models, which describe such processes, may

consider huge amounts of particles: up to billions of them, and even more. In addition, each particle can be described by dozens of parameters and the volume of output data in such CE can be estimated in terabytes.

Processing of such volumes of data in serial mode can potentially take years, and optimization of computing code does not bring a significant acceleration of the computations. Therefore, currently the most widely used approach to accelerate the large-scale computing is it's paralleling, which means that a great number of compute nodes would process a large amount of data each handling apart of it.

As a result of paralleling, each node receives only a small part of the data set which is easy to manipulate with.

This technique significantly reduces the time required to complete data processing, but leads to several problems concerning the data storage. Most often, after performing calculations compute nodes exchange the results of computations, and master process assembles them in RAM or in a storage device as one large array or a file. However, in the large-scale computations the size of the result array (file) can significantly exceed the resources of the master node. In this case, each compute node stores the results in isolation. The last described method of storage has several advantages. The first one is the lack of need to sequentially read all the results for further processing (for example, for visualization purpose) because each computational node only reads it's part of the data. The second advantage is that each individual data file is typically not very large (compared to the full data set), and thus it takes less processing time. Such data can be reached in various ways, for example using a distributed file system, on-the-node-process reading, or using the applications allowing to send data over the network, such as the SFTP.

The scientific programs that store data in the form described above, are considered in this article. The results of the simulation based on the algorithm, described in the article [1] were used as a data set for studying parallelization capabilities of the developed "mmdlab" library.

One of the ways of CE data representation is a two- and three-dimensional visualization.

In order to assemble a complete state of the simulation results, it is required to read and process the data from each compute node, which in itself is a resource-intensive task. In most cases, the calculated data formats and storage methods differ depending on the calculation program. Therefore, such programs usually have their own visualizer, and calculate all the necessary visualization data in the process of computation, collecting them on the master node. In this case, the visualization is provided by the means of such programs (LAMMPS, and others). Another way is to save data in the well-known standardized containers (HDF5, VTK, and other), which are supported by the majority of software for scientific visualization. The problems of such methods of storage and rendering are the limited possibilities of the used visualization software in regards to visualization and post-processing, and in the case of well-known standards of data storage there occurs the problem of loading large files.

This paper presents an attempt to create a flexible tool that allows importing, processing and visualization of data from different sources, regardless of it's structure: whether the data is in known formats or distributed calculation results in a custom format.

The results obtained using the computer program described in the article [1] were considered as a test case.

In view of the parallel algorithms and storage features, this data can be a one big file that describes the general state of the simulated system, as well as a distributed data, processed by every computational process separately. The results obtained from the simulation are the information about the interactions of the gas molecules with the metal atoms near the surface. This process is characteristic for many technological microsystems used in nanotechnology.

2. Problem Statement

The problem of collecting and processing the distributed data obtained as a result of some calculation program has several key features.

Firstly, it is the specifics of the problem domain. As a result of searching among the various simulation packages, there has not been found suitable means for parallel loading of distributed data relating to the considered task. This problem drove us to do this research.

Secondly, the scale of the input data can differ greatly. It can be a small one-dimensional array or a large number of files distributed across the various computational nodes and file systems. Such problems are usually solved by means of a software system that generated this data, or by development of a specialized "loader" tool, which understands input-output formats used by the calculation program.

Thirdly, there is a need to process such results for convenient representation on charts or in 3D visualization.

Due to the features described above, in this work we made an attempt to create a framework for the software complex with the following features:

- Parallel reading of data from different sources;
- User-defined data formats support;
- Custom data filters and processors support;
- Data visualization solution;

It is important to emphasize that in the case of development of such library its expandability has a significant role. It should be relatively easy to use the developed framework for processing the data stored in any format, and to integrate it with the other known solutions for visualization and data processing. As the initial stage of development we chose the problem of post-processing and visualization of the results obtained in work described in the article [1].

This task involves the consideration of all the listed features of the selected application, because of the distributed structure of the data in different computer systems with remote access to it via SSH.

3. Development tools

There are many known solutions for task-based paralleling and data visualization.

Feature of these solutions is the difficulty of their use, setup and installation.

Among the known solutions for clustering can be noted Apache Hadoop. This is a large and complex solution, which implements MapReduce model for task-based parallel processing. However, for the considered problem, it has many unnecessary features, such as a distributed file system (HDFS) and requirement of installation on computational nodes.

For general scientific visualization, there is a variety of software packages, for example, Paraview, VMD, Tecplot. Each of these software packages has its own format of data storage, and is also able to read the standardized formats. However, in the case of a custom data format or a complex data distribution all of these solutions require implementation of a special data loader.

Taking all the above into account, we decided to add into the developed library the support of the integration into such packages, and its own visualization and clustering tools. Furthermore, "mmdlab" library has a minimum set of dependency and does not require installation on the compute nodes.

In view of the need for the above-mentioned integration into well-known solutions, as well as the requirements posed by the expandability of developed framework, we decided to use an interpreted programming language Python, due to the fact that almost all of that packages use Python in their plug-in systems.

3.1 Python

Python [2] is a widely used in scientific community general-purpose high-level programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than it would be possible in languages such as C++ or Java. The syntax of kernel of Python is very simple and short, at the same time a standard library gives the large volume of useful functions and convenient data structures. It is also a cross-platform, so you can use it (with some restrictions), both under the MS Windows and Linux operating systems.

Python supports multiple programming paradigms, including object-oriented, imperative and functional programming or procedural styles. It features a dynamic type system and automatic memory management, full introspection, exceptions and multiprocessing.

The developers community created a lot of computer science libraries, that makes Python one of the most commonly used languages for big data analysis and scientific calculations.

Though Python already has version 3, in this study we used Python version 2.7, in view of the fact that some used libraries (for example, Mayavi2) were written in Python 2.7, and Python 3 and Python 2.7 in some cases do not have backward compatibility.

3.2 IPython

IPython [3] is an interactive shell for Python language, which adds an expanded introspection, additional command syntax, code highlighting and autocomplition. The main feature of this project is that it provides the core for Jupyter web-application, which allows to write scripts in Python, R, and BASH directly in the browser, as well as interact with the objects of visualization. In this work IPython notebook application has been selected as the web-control system.

3.3 Accelerators of computations

Despite all the advantages of the main realization of the interpreter CPython, it is necessary to remember that the Python is a high-level interpreted programming language. It cannot provide high performance itself, due to the memory management system and dynamic typification. It is very easy to use, but if performance is critical it is necessary to implement CPU-critical code in C or C++, to avoid the overhead of interpreter calls. However, there are several technologies allowing to evade the low-level programming.

Another big disadvantage of the CPython interpreter is associated with the speed and performance in multithreading. The last is caused by use of the GIL (Global Interpreter Lock) mechanism representing mutex (the elementary binary semaphore) which is not allowing different threads to process the same bytecode at the same time. Unfortunately, this lock is necessary, since the memory management system in CPython is not thread-safe. The following methods were considered to avoid this limitations.

3.4 Numpy

Numpy [4] is an open source library for Python. It implements fast multi-dimensional arrays and plenty of parallel (vectorized) algorithms for linear algebra, Fourier transform and other applications. Since Numpy is written in C, the executable code of the library is compiled into native code, and there is no need for its interpretation, gaining significant speedups of the array-processing methods. The threads that run inside Numpy do not depend on the GIL, present in the CPython, and therefore its use accelerates the execution of algorithms by parallelization. Besides Numpy has detailed documentation that facilitates the development and maintenance of the software. All these features make Numpy reasonable choice for array processing in Python.

3.5 Numba

Numba [5] is optimizing Just-In-Time (JIT) compiler, which allows to accelerate the time-critical code by compiling it into native code. Unlike Cython, Numba does not require explicit type annotations (but supports it) and does not translates the code in C language, which simplifies the use of this technology. In order to show Numba which methods are needed to be optimized, the user must use the simplest means of Python language, called a decorator.

Marked by the special decorator methods Numba optimizes and compiles to machine code using LLVM (Low Level Virtual Machine) infrastructure. With the ability to turn off the GIL, as well as the compilation to native code without using the Python C API (for the methods that operates elementary types), Numba compiler can generate more efficient and optimized bytecode. Numba also automatically vectorizes all that it can handle, utilizing the capabilities of multiprocessor systems to the maximum.

```
from numba import jit
@jit(nogil=True, nopython=True)
def numpy_numba_func(vx, vy, vz, multiplier=100, divider=3.0):
    return multiplier*((vx*vx) + (vy*vy) + (vz*vz)) / divider
def numpy_func(vx, vy, vz, multiplier=100, divider=3.0):
    return multiplier*((vx*vx) + (vy*vy) + (vz*vz)) / divider
```

Listing 1. Numba and Numpy array multiplication.

Table 1. Numba and Numpy performance comparison.

N	Numpy	Numba	Speedup
10^6	0.19 ms	0.07 ms	2.77
10^7	1.62 ms	0.74 ms	2.19
10^8	16.06 ms	7.4 ms	2.17

Table 1 compares the speed of execution of the same Python code (multiplication arrays with multiplying and dividing by a constant, see Listing 1), in one case without Numba, in the other using this technology. Testing was performed on a system with the Intel Core I7-3630QM CPU.

It should be noted that the algorithm shown in Listing 1 is not parallel in the means of code, and the vectorization is performed by Numpy.

The Table 1 shows that Numba allows to speed up the execution nearly twice due to JIT compilation, without any special optimization, such as, most likely, would be needed while using any other tools, such as Cython.

4. Parallelization tools

Considering a GIL mechanism, presenting in CPython, the use of standard Python threads is not an effective solution for parallel processing. GIL does not allow multiple threads run simultaneously on different cores (within one interpreter process) even on a multiprocessor system. However, running multiple processes of interpreters, which can exchange data, completely solves this problem. The only distinctive in this case is that the launch of the process is a much more prolonged operation than starting threads, and usage of multi-process application on small data is not rational. There are several tools for easy management of such tasks.

4.1 Multiprocessing

Multiprocessing [2] is a standard library module that provides an interface to create and manage multiple interpreters processes. Its API is similar to the threading module of the standard library. It also adds some new features, such as the Pool class, representing the abstraction and control mechanism for a set of parallel interpreter processes. Multiprocessing also implements interprocess primitives, such as queue and mutex. It is also worth noting that each process of the interpreter works in separate memory space, therefore there is no need to worry about race conditions when writing or reading variables, unless they are declared as an object in shared memory. Communication between the processes of the interpreter within a given library is through interprocess communication channel, based on pipes, using the pickle module, allowing to "serialize" and "deserialize" the Python objects (serialization - the process of transferring any data structure into a bit sequence; deserialization - the restoration of the initial state of the data structure from a bit sequence). All the tasks of synchronization and object transferring are carried out by the Multiprocessing module. Therefore, the user does not need to solve the problem of confirming that all data used in the calculation has been updated.

4.2 ParallelPython

ParallelPython (PP) [6] is a library used to solve the problem of clustering applications. Its implementation has a client-server structure and it requires installation of the server part on the compute nodes. However, the server program of the PP is a simple one-file script, that can be transferred into the node in any possible way. Because of the simplicity of PP interface, it allows to run a computational task on a parallel cluster in few lines of code.

This library has its own load balancer, and it also monitors the status of nodes and redistributes tasks in case of non availability of one of them. With Multiprocessing module, ParallelPython allows simply and conveniently use all of the capabilities of the cluster computing.

Listing 2 shows an example of summing up the plurality of arrays in parallel mode, using ParallelPython and Multiprocessing.

```
import pp
import numpy as np
ppservers = ("10.0.0.1","10.0.0.2","10.0.0.3","10.0.0.4")
serv = pp.Server(ncpus = 2, ppservers=ppservers)
def mpsum(array):
    pool = multiprocessing.Pool(2)
    half = len(array)/2
    s = sum(pool.map(sum, [array[:half], array[half:]]))
    return s
arrays = [np.ones(5000) for i in xrange(10) ]
imports = ("multiprocessing",)
depfuncs = tuple()
jobs = [serv.submit(mpsum,(a,), depfuncs, imports) for a in arrays]
s = sum([job() for job in jobs])
print s
```

Listing 2. Parallel Python and multiprocessing usage for multiple arrays summation.

At every computational node, two processes start by ParallelPython and each of them starts other two process by means of Multiprocessing. It is worth noting that this library, as well as Multiprocessing, uses the "pickle" module to serialize data and tcp / ip network messaging.

5. Visualization tools

As it was already mentioned, there are many third-party tools for data visualization. The "mmdlab" library presented in this work can be used as a tool for preparation of data for the visualization in such packages, however it was also decided to add its own visualization capabilities. During the research it has appeared that the listed below libraries almost do not concede in options to the well-known packages for scientific visualization.

5.1 Mayavi2

Mayavi2 [7] is a Python framework, which allows to build a general-purpose scientific visualization. It gives user a possibility to load and render the data in a separate GUI application and also has a convenient Python API for scene construction and rendering. This library is built over the well-known in scientific community VTK library.

Mayavi2 gives ample opportunities for the visualization of data, beginning from hydrodynamic calculations and finishing with atomistic data. In the case of the interactive GUI mode, tools for changing the rendering parameters, such as the size of objects, color schemes, filter settings are also available. Mayavi2 also has a possibility of the offscreen-rendering (without displaying image), that is extremely important for the server, distributed and batch operation of a large number of data.

```
import pp
import numpy as np
ppservers = ("10.0.0.1","10.0.0.2","10.0.0.3","10.0.0.4")
serv = pp.Server(ncpus = 2, ppservers=ppservers)
def mpsum(array):
    pool = multiprocessing.Pool(2)
    half = len(array)/2
    s = sum(pool.map(sum, [array[:half], array[half:]]))
    return s
arrays = [np.ones(5000) for i in xrange(10) ]
imports = ("multiprocessing",)
depfuncs = tuple()
jobs = [serv.submit(mpsum,(a,), depfuncs, imports) for a in arrays]
s = sum([job() for job in jobs])
print s
```

Listing 3. KDE calculation and visualization script using SciPy and Mayavi.

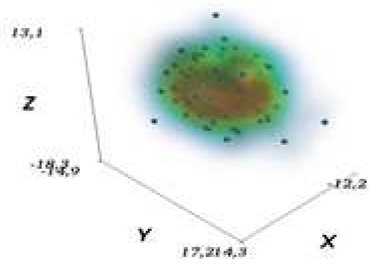

Fig. 1. Listing 3 execution result: Kernel Density Estimation as volume visualization.

Listing 3 and the Fig. 1 show an example of the density distribution calculation of points and its three-dimensional visualization using Mayavi2 and library for scientific computing SciPy.

5.2 Matplotlib

Matplotlib [8] is a Python library for building high-quality two-dimensional graphs. It is widely used in the scientific community. Usage of Matplotlib is very similar to the usage of the plot methods in MATLAB, however, they are independent projects.

It is particularly convenient that the plots, which are drawn with the help of this library can be easily integrated into applications written with different libraries for GUI construction. Matplotlib can be integrated into applications written using the wxPython, PyQt and PyGTK libraries.

Matplotlib module is not included in the standard library, but it is the de facto standard for the visualization of numerical information.

6. Distributed data access

The data obtained from the algorithm, described in the article [1] has distributed structure, and is stored on the compute nodes, used for simulation.

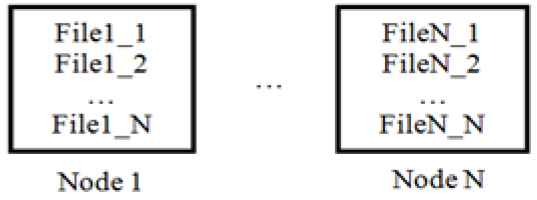

Fig. 2. Data distribution structure.

Fig. 2 shows an example of such data arrangement. The composition of all the files is a complete form of the system simulated by means of molecular dynamics. It happens that the computational nodes use the shared disk space, for example, by means of the NFS (Network File System). However, access to the data from the client-side which needs to read and process the data is open only via SSH. Paramiko library can be used to solve this problem.

6.1 Paramiko

Paramiko [9] is a library for the Python language, which provides implementation and interface for interacting with remote systems via SSHv2 protocol. This library has both client and server implementations. In addition, Paramiko provides a convenient API, which implements objects of "file" type, which are representing files on the remote filesystem. This functionality was used as a basis for the implementation of SSH collector in the represented work.

7. Implementation details

Using the tools above, there was initiated the development of the software complex, allowing to achieve the objectives, namely the parallel data reading and processing, as well as their visualization. As an initial stage, "mmdlab" package was written which implements a general purpose API for such tasks. Below are described the implementation problems we have to handle, application and solutions with the means of the developed library. There is also drawn further attention to the implementation peculiarities in some parts of the package.

7.1 Parallel data access

A module for reading and partial processing of the input data was named "datareader". In this module have been implemented the necessary objects for reading and representation of the data, such as Container, Parser and means of access to the files on the local file system and via SSH. In the terminology of "mmdlab" package, Container is a structure that stores the read data in a user-defined format. Parser is a special object that reads binary data structure and parses them, thereby obtaining a container. The Parser class receives the raw data from the Transport object that provides an interface for the access to the local or remote file system.

Inheriting and combining objects from these classes, the user can easily make the loader, that parse a custom data format, and accesses it using any protocol, such as SSH or HTTP.

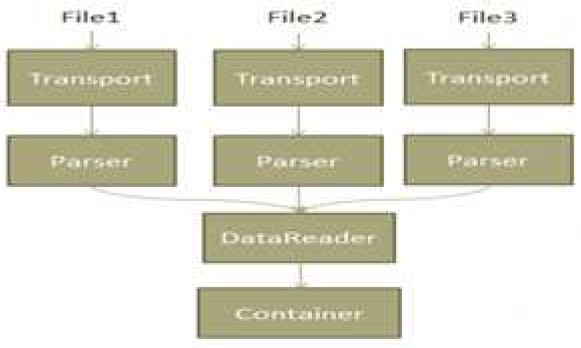

Fig. 3. MMDLAB components scheme.

On the Fig. 3 are shown the "mmdlab" components interactions.

Let's consider the reading procedure of the MD system's particular state described the article [1]. Given the distributed structure of input data, a single state of the system is a set of files of the atomistic data. For each of them it is necessary to read, parse and compile binary structure into a single container that contains the representation of the simulated system. For the performance needs it is necessary to use a parallel algorithm for the reading and processing of the data.

Master process launches N slave-processes that are able to load and parse the data.

Then it begins to give every data file address to a every free process. When the slave process has finished the reading and parsing procedure, and assembled its part of the container, the master process combines the loaded data with its master container, and then assigns a new file to the slave process. After all the slave processes are completed, and there are no more files for reading, master process provides the necessary post-processing for the container, where all of the available data is stored, and sends it to the next data processor in line. It should be noted that in some cases it is not necessary to send all the data to the master host. For those cases, the "mmdlab" supports a possibility to use the post-processing pipeline in the slave processes, so they can make necessary calculations and send back only the result, but not all the processed data set.

In order to enhance the ability of "mmdlab" package for reading the custom-format data, it is required to describe the new entity for storage and loading of such data.

As an example, consider the implementation of such entities for reading a CSV (Comma-Separated Values) format with three columns.

```
class CsvCtr(dr.containers.DummyContainer):
    def __init__(self):
        self.cols = [[],[],[]]
    def append_data(self,data):
        for i,d in enumerate(data[:]):
        self.cols[i].extend(d)
class CsvParser(dr.parsers.DummyParser):
    def data(self):
        cols = [[],[],[]]
        for line in self.transport.readlines():
            c = line.split(",")
            for i in range(0,3):
                cols[i].append(c[i])
        return cols
nodes = \
({"ip":"10.0.0.1","pwd":"123","login":"test"},
 {"ip":"10.0.0.2","pwd":"123","login":"test"})
remotedirs = [(sys.argv[1], node) for node in nodes]
transport = dr.transport.RemoteDirs(remotedirs)
parser = CsvParser()
rdr = dr.DistributedDataReader(file_mask="1*.csv",
                              transport=transport, parser = parser,
                              container = CsvCtr)
container = mmdlab.run([rdr, ])
```

Listing 4. CSV Container and Parser implementation using "mmdlab" package.

Listing 4 shows an example of such an extension to CSV reading from remote file systems via SSH.

In practice the user will need to describe the new class inherited from the class DummyContainer and to redefine the append_data method in it. Also it will be required to describe the class for raw data parsing.

7.2 Pipeline

In this work, to run reading and processing tasks, it is proposed pipeline-type interface (see Listing 5, the mmdlab.run part).

This method makes it possible to run an execution of a chain of actions in one line, each of which is carried out over the result of the previous task. Also parallel operations over the same result of the previous method are supported.

For example, the call of mmdlab.run([generate, [f1, f2, f3], sum]) first performs the "generate" method, then in parallel mode it runs three processes: "f1", "f2", "f3" each operating on the result of the "generate", and in the end it will summarize the obtained values. Restrictions on objects in the pipeline are simple: the object has to be callable, it should take the data for processing as an argument and it should return an object.

```
from mmdlab.datareader.shortcuts import read_distr_gimm_data
import mmdlab
import sys
reader = read_distr_gimm_data(sys.argv[1],sys.argv[2])
filter_reg =
mmdlab.dataprocessor.filters.RegionFilter([0,10,0,10,0,10])
parts_descr = \
{ "Nickel" : { "id" : 0, "atom_mass" : 97.474, "atom_d" : 0.248}, \
"Nitrogen" : { "id" : 1, "atom_mass" : 46.517,"atom_d" : 0.296} }
filter_split = mmdlab.dataprocessor.filters.SplitFilter(parts_descr)
container = mmdlab.run([reader,  filter_reg, filter_split ])
met,gas = container["Nickel"], container["Nitrogen"]
mp = mmdlab.vis.Points3d(met, scalar=met.t, size=met.d,
                                          colormap="black-white")
gp = mmdlab.vis.Points3d(gas, scalar=gas.t, size=gas.d,
colormap="cool")
mmdlab.vis.colorbar(gp, "Gas T")
mmdlab.vis.show(distance=20)
```

Listing 5. Reading, processing and visualization of the atomistic data using "mmdlab" package.

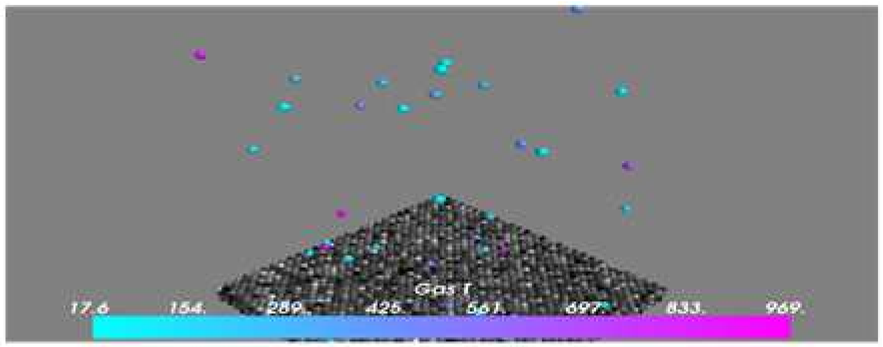

Fig. 4. The result image produced by execution of Listing 5.

At the current stage of development, when you run a multithreaded processing over the previous action the result will be copied to each of the child process.

In the future we plan to add some additional entities, allowing to manage the execution workflow, such as a special object that allows to perform an action in the master process, and to send the result's parts to the slave-processes. This may be necessary, for example, for the separation of the array into a multiple parts, and process each in a separate slave-process without sending the entire array to it.

Due to the fact that the pipeline is implemented by means of the interface module Multiprocessing, consider some of the problems encountered.

7.3 RAM leak in parallel processing

Let's consider the reading procedure of the DistributedDataReader class (see Listing 6).

```
class DistributedDataReader:
    ...
    def read(self):
        ...
        files = self.transport.list(self.file_mask)
        container = self.container()
        pool = Pool(processes=self.np, maxtasksperchild=self.mtpc)
        results = [pool.apply_async( rd, \
                      args=(f,self.transport.filer(),self.parser))\
                      for f in files]
        for ct in results:
            container.append_data(ct.get())
        return container.finalize()
```

Listing 6. A part of DistributedDataReader class.

During the testing it was found that a resources leak appears in the multiprocessing mode. After starting the pool of processes, and performing a variety of tasks in it, memory consumption increases dramatically. It became apparent that by default the started by Multiprocessing library interpreter processes handle all the scheduled tasks without restarting.

Each task which is carried out in such processes leaves the context, which becomes bigger in the volumes of consumed memory as the more data the task returns. As a result, after long-term execution of multiple tasks at the computational node the RAM came to an end.

The proposed solution of this problem is as follows. The object of a processes pool has a special parameter of the constructor named "maxtaskperchild", allowing to set the number of tasks that a single interpreter process can handle. When the counter of finished jobs becomes more then this value, the master-process algorithm will restart the interpreter. Changing this parameter allows to vary the maximum amount of memory consumed. However, it should be noted that the smaller the value, the more often the master process will restart child processes' interpreters. It can take noticeable amount of time.

Within the considered task of processing large amounts of data, the time is not critical, and installation of rather small value is quite justified because of memory limits.

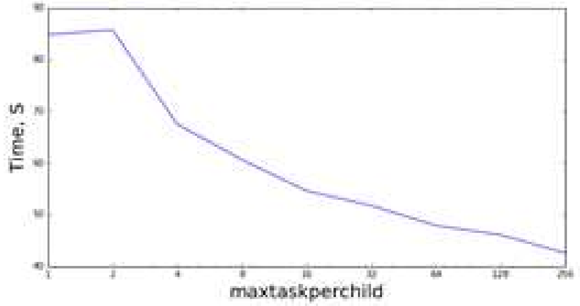

Fig. 5. Loading time of 256 files depending on a "maxtaskperchild" parameter, logarithmic scale.

Fig. 5 shows the dependence of the loading time on the "maxtaskperchild" parameter. The loader uses multiprocessing module, with the pool consisting of one process, and loads 256 data files in serial mode.

Taking into the account the Fig. 5, the optimal behavior of the processes pool is to restart the slave-workers every 16 tasks. It makes possible limiting the consumption of RAM and at the same time keeps the overhead of the interpreter restart time influence almost negligible.

7.4 Multiprocessing and Pool of Pools

Another problem encountered in the development process is the fact that the default multiprocessing library does not allow to create "nested" pools for processes.

In particular, if there appears a necessity to run in parallel the processes of reading a plurality of states of the studied system (this will start new slave-processes that should start a lot of reading processes), for example, for the particles' trajectories construction, so the Multiprocessing module will not allow to do it. The introspection which is supported by the Python language fully helps with the solution of this problem.

The "mmdlab" package developed in this work has a construction shown in Listing 7 included in it. It redefines the _get_daemon and _set_daemon methods at the "multiprocessing.Process" class and provides a new object, inherited from the Pool class. It should be used instead of the standard Pool class from Multiprocessing module.

```
import multiprocessing
import multiprocessing.pool
class NoDaemonProcess(multiprocessing.Process):
    def _get_daemon(self):
        return False
    def _set_daemon(self, value):
        pass
    daemon = property(_get_daemon, _set_daemon)
class MultiPool(multiprocessing.pool.Pool):
    Process = NoDaemonProcess
```

Listing 7. MultiPool class, allowing to run pool of processes inside child process, created by multiprocessing module.

7.5 Data processing

For processing and filtering data in developed "mmdlab" library the same mechanisms as for the data reading are used. The so-called "pipeline" architecture is used which implicates the container object passing through a chain of a great number of data processors, that can change, supplement a container or create a new one. The "run" method in the "mmdlab" package passes the container obtained from the previous task to the input of the next processing method. The implementation of these processing methods can be both serial and parallel.

In the application to the analysis specific objective of molecular dynamics simulations' results from the article [1], the objects for data post-processing have been added to the developed library. For example, a filtration of particles by various criteria, in particular for getting the particles only from specified area, for filtration by indexes and division of particles according to physical materials.

All computationally intensive procedures were optimized by using Numpy and Numba.

As a simple example, let's consider the task of visualizing of the particles' position and temperature that are divided by criteria of physical material in the predetermined area. Such problem can be solved using "mmdlab" library in the following way (see Listing 5). First, the user creates an object of the data loader, setting their location in the filesystem and a time mark.

Then they need to specify the description of particles, which the division filter will work with, and create the corresponding objects of filters (the location filter and the division filter). Lastly they need to pass these objects to the pipeline. Calculation of temperature is performed during the container's post-processing stage.

Listing 5 and Fig. 4 show the listing of such task and the execution results.

```
import sys
import mmdlab
from scipy.stats import *
from mmdlab import parallel
from mmdlab.datareader.shortcuts import *
from mmdlab.dataprocessor.filters import *
rdr = read_distr_gimm_data(sys.argv[1],0)
def calc_kde(kde,  data):
    return kde(data.T)
parts_descr = { "Nickel" : { "id" : 0}, "Nitrogen" : { "id" : 1}}
filter_split = SplitFilter(parts_descr)
filter_reg = RegionFilter([0, 100, 0, 100, 0, 100])
cont = mmdlab.run([rdr, filter_reg, filter_split])
gas = cont["Nitrogen"]
kde = gaussian_kde(np.vstack([gas.x,gas.y,gas.z]))
xi, yi, zi = np.mgrid[0:gas.x.max():30j,
                      0:gas.y.max():30j,
                      0:gas.z.max():30]
c = np.vstack([item.ravel() for item in [xi,yi,zi]])
cores = sys.argv[2]
nodes = ("192.168.6.15","192.168.6.20")
cluster = parallel.Cluster(nodes)
args = [(kde,a) for a in np.array_split(c.T, cores)]
cluster.map(calc_kde, args)
density = np.concatenate(results).reshape(xi.shape)
```

Listing 8. KDE Clustering example using "mmdlab" package.

7.6 Cluster processing

For testing of the cluster mode was used the combination of the master node with the Intel Xeon E5-2650 (32 cores) and 6 compute nodes (Intel Xeon 5150 2.66 GHz, 24 cores) with shared file system over NFS. It gives certain freeness in respect of access to the data: it is not required to associate the input and the node on which processing is started, as any datafile is available from any of nodes.

However, such configuration has a bottleneck: the storage input-output performance. As a result, it was decided to use the following strategy: a master node, which is a physical data storage, in the multiprocess mode loads data into memory and sends it to the cluster nodes in the form of internal representation, without the data processing. In contrast to the strategy of "reading on each node" the described way allows to use the computational capabilities of the subordinated nodes on maximum, with the minimum input-output waiting, maximizing disk input-output utilization.

In case of difficult visualization for which processing and rendering takes more time than reading one system state, such approach allows to reduce the average time of full processing almost to the data reading time, which is the potential minimum time of processing.

As an example of clustered task, consider the problem of constructing three-dimensional field of the gas density in the computational domain using Kernel Density Estimation (KDE) algorithm, implemented in SciPy library (see Listing 8).

The graphs of execution time (see Fig. 6) and the acceleration (see Fig. 7) of such calculations, depending on the number of processors for a variable number of subtasks are shown below.

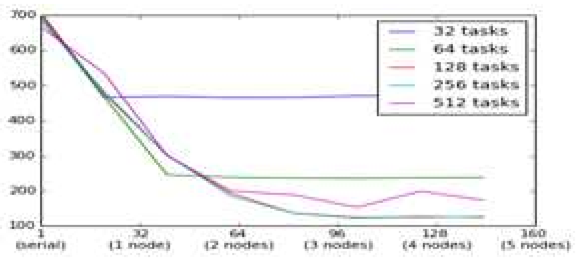

Fig. 6. Processing time for parallel KDE algorithm with various number of subtasks, depending on the number of used processors.

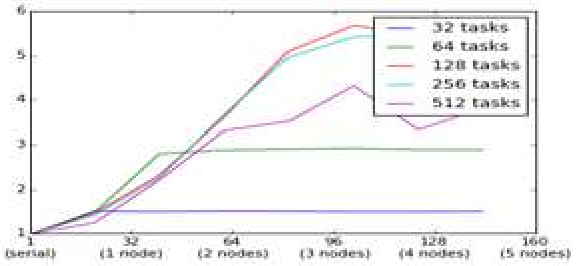

Fig. 7. Speedups for parallel KDE algorithm with various number of subtasks, depending on the number of used processors.

It should be noted that if the number of tasks is less than the number of master node processes (which is up to 32), then the increasing of the process's count in this calculation is not effective. Also, the acceleration increases with the number of nodes involved in the computation, rather than with the number of actual processes. This is due to the following two features:

- PP considers that the overhead of process start-up and data transfer is significantly less on the master-node, than on the slave-nodes. Thus, it loads the master node to the maximum, before it starts to send jobs to the slave-nodes;
- Numpy already vectorizes array operations over all available cores, and the addition of a new processor will not make a significant acceleration;

Also we need to note that the PP, which is used as a library for clustering, automatically distributes the load across nodes, depending on the tasks execution time. So it makes sense to divide the original problem into a number of subtasks more than the number of available processes, if there are some "weak" nodes in the cluster. In this case PP forms a queue and gives tasks to the nodes taking into account efficiency of each node, thereby providing a load balancing.

7.7 Visualization

For the visualization in this work Mayavi2 and Matplotlib library were used. For convenient usage of the common rendering methods, the "mmdlab.vis" module was included, which is a wrapper over the methods of these libraries, combining their capabilities to achieve the desired result. Due to the single-threaded architecture of Mayavi and Matplotlib, data visualization process is currently supported only in the single-threaded mode within a single process. However, "mmdlab" allows to run a hybrid task of reading and rendering on a set of nodes and in the multiprocess mode, which significantly accelerates the rendering of frame-by-frame video animations.

For example, consider the task of rendering an animation, which consists of frames representing the state of the studied system in consecutive timepoints. Basic data can be distributed across the multiple nodes, thus the visualization can be run on each of the nodes, and then the result can be collected on the master-node. The following algorithm is proposed for the solution of such a problem:

- On each of the specified nodes run a sequence of reading and visualization;
- Collect all the frames that were drawn on the master node;
- Assemble an animation from collected frames;

To build an animated GIF format file "mmdlab" library uses the program "convert" from the ImageMagick [10] utils.

8. Conclusion

This paper presents the experimental version of a high-level library "mmdlab" for the Python language. Usage of such library makes it possible to perform a simple clustering and paralleling for the various types of processing tasks, such as reading, post-processing and visualization.

It can operate over the large-scale data, distributed over the computational nodes in parallel mode.

The main tasks of the development of this library are the analysis and visualization of the data obtained as the result of MD simulation of gas-metal microsystem described in the article [1]. To achieve this goals it was necessary to process about 1.3 TB of data obtained from one simulation, and there were three simulations with different materials temperatures. Usage of the "mmdlab" library allowed to closely observe the effect of nitrogen adsorption on a nickel plate (see Fig. 8) including an analysis of the individual particles' trajectories.

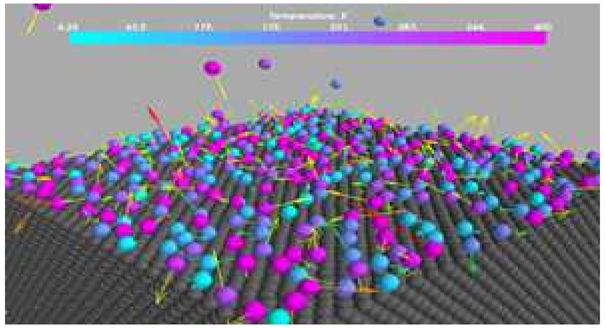

Fig. 8. Adsorbtion of nitrogen on nickel plate and particle trajectory visualized using the "mmdlab" package.

Special attention was paid to a possibility of extension of the library. It is possible thanks to flexibility of the used tools. As a result, usage of the developed library can be extended to reading and visualization of potentially any structures of data.

9. Acknowledgment

Work is performed with assistance of the Russian Foundation for Basic Research (grants No. 15-07-06082-a, No. 15-29-07090-ofi_m).

10. References

[1]. V.O. Podryga, S.V. Polyakov, D.V. Puzyrkov, “Supercomputer Molecular Modeling of Thermodynamic Equilibrium in Gas–Metal Microsystems” (in Russian), in Vychislitel'nye Metody i Programmirovanie [Numerical Methods and Programming], vol. 16, no. 1, pp. 123-138, 2015 (in Russian).
[2]. Python official documentation. 04, Feb. 2016, https://www.python.org/
[3]. P. Fernando, E.G. Brian, “IPython: A System for Interactive Scientific Computing” (in English), in Computing in Science and Engineering, vol. 9, no. 3, pp. 21–29, 2007. (2015, Feb. 4), [Online]. Available: http://ipython.org
[4]. Stéfan van der Walt, S. Chris Colbert and Gaël Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation, Computing in Science & Engineering, 13, 22-30 (2011), DOI:10.1109/MCSE.2011.37
[5]. Numba official documentation, 04, Feb. 2016, http://www.numba.pydata.org/
[6]. Vanovschi V., Parallel Python Software, http://www.parallelpython.com
[7]. Ramachandran, P. and Varoquaux, G., `Mayavi: 3D Visualization of Scientific Data` IEEE Computing in Science & Engineering, 13 (2), pp. 40-51 (2011)
[8]. John D. Hunter. Matplotlib: A 2D Graphics Environment, Computing in Science & Engineering, 9, 90-95 (2007), DOI:10.1109/MCSE.2007.55
[9]. Paramiko official documentation, 04, Feb. 2016, http://www.paramiko.org/
[10]. ImageMagick official documentation, 04, Feb. 2016, http://www.imagemagick.org/

Параллельная обработка и визуализация для результатов моделирования методом молекулярной динамики

Д. В. Пузырьков <dpuzyrkov@gmail.com>
В. О. Подрыга <pvictoria@list.ru>
С. В. Поляков <polyakov@imamod.ru>
Институт Прикладной Математики им. М. В. Келдыша Российской Академии Наук
125047, Москва, Миусская пл., д.4

Аннотация. В этой работе авторами представляется библиотека "mmdlab" для интерпретируемого языка программирования Python. Эта библиотека позволяет осуществлять чтение, обработку и визуализацию результатов численных расчетов задач молекулярного моделирования. Учитывая большой объем данных, получаемый в результате проведения таких симуляций, существует необходимость в параллельной реализации алгоритмов для обработки таких объемов. Параллельная обработка должна выполняться как на многоядерных системах, таких как обычный современный компьютер, так и на суперкомпьютерных системах и кластерах, где происходило численное моделирование методом молекулярной динамики. В процессе разработки данной библиотеки была изучена эффективность языка Python для таких задач и были рассмотрены инструменты, позволяющие увеличить производительность программ на этом языке. Также были изучены возможности данного языка в отношении параллельных вычислений и инструменты, позволяющие использовать для вычислений системы кластерного типа. Кроме того, были исследованы проблемы загрузки и обработки данных, расположенных на множестве вычислительных узлов. Это было вызвано необходимостью обрабатывать данные, полученные с помощью параллельного алгоритма, который выполнялся на нескольких вычислительных узлах и сохранял результаты на каждом из них. В качестве инструмента для научной визуализации был выбран пакет с открытым исходным кодом "Mayavi2". Разработанная библиотека "mmdlab" была использована для анализа результатов МД моделирования взаимодействия газа с металлической пластиной. В результате применения данной библиотеки удалось в деталях наблюдать эффект адсорбции, который важен для многих практических приложений.

Ключевые слова: параллельная обработка; визуализация; молекулярная динамика; Python; Mayavi2

DOI: 10.15514/ISPRAS-2016-28(2)-15

Для цитирования: Пузырьков Д.В., Подрыга В.О., Поляков С.В. Параллельная обработка и визуализация для результатов моделирования методом молекулярной динамики. Труды ИСП РАН, том 28, вып. 2, 2016 г., стр. 221-242 (на английском). DOI: 10.15514/ISPRAS-2016-28(2)-15

Список литературы

[1]. Подрыга В.О., Поляков С.В., Пузырьков Д.В., Суперкомпьютерное молекулярное моделирование термодинамического равновесия в микросистемах газ-металл. Вычислительные методы и программирование, том 16, no. 1, стр. 123-138, 2015.
[2]. Python official documentation. 04, Feb. 2016, https://www.python.org/
[3]. P. Fernando, E.G. Brian, “IPython: A System for Interactive Scientific Computing” (in English), in Computing in Science and Engineering, vol. 9, no. 3, pp. 21–29, 2007. (2015, Feb. 4), [Online]. Available: http://ipython.org
[4]. Stéfan van der Walt, S. Chris Colbert and Gaël Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation, Computing in Science & Engineering, 13, 22-30 (2011), DOI:10.1109/MCSE.2011.37
[5]. Numba official documentation, 04, Feb. 2016, http://www.numba.pydata.org/
[6]. Vanovschi V., Parallel Python Software, http://www.parallelpython.com
[7]. Ramachandran, P. and Varoquaux, G., `Mayavi: 3D Visualization of Scientific Data` IEEE Computing in Science & Engineering, 13 (2), pp. 40-51 (2011)
[8]. John D. Hunter. Matplotlib: A 2D Graphics Environment, Computing in Science & Engineering, 9, 90-95 (2007), DOI:10.1109/MCSE.2007.55
[9]. Paramiko official documentation, 04, Feb. 2016, http://www.paramiko.org/
[10]. ImageMagick official documentation, 04, Feb. 2016, http://www.imagemagick.org/