Parallel Computing is Everywhere
S. Bassini et al. (Eds.)
IOS Press, 2018
© 2018 The authors and IOS Press. All rights reserved.
doi:10.3233/978-1-61499-843-3-242

CalCul: A Python-Based Workspace for High-Performance Parameters-Sweep in Scientific Legacy Codes

Gal OREN[a,b] and Guy MALAMUD[b]
[a] *Department of Computer Science, Ben-Gurion University of the Negev, P.O.B. 653, Be'er Sheva, Israel*
[b] *Department of Physics, Nuclear Research Center-Negev, P.O.B. 9001, Be'er-Sheva, Israel*

Abstract. Scientific legacy codes are usually holding large numbers of physical arrays which have been used and updated by the code routines, and the code parameters are set using a simple textual data file. Therefore, in cases when there is a need to perform a parallel large-scale parameters-sweep, the user needs to manually replicate the code multiple times, change the data file to its needs, run the code and control its performances - at each working directory separately. This task is practicable when there is a need to create several computations, but impossible when there is a need to run a large-scale parallel parameters-sweep on a cluster with thousands of calculations. In this work we present CalCul - A Python-based Workspace for High-Performance Legacy Scientific Codes, which is able to automatically render most of legacy codes data files into a python object, allowing the user – using many other functionalities – to control and handle all the needs of a parameters-sweep using the simplicity and sophistication of Python libraries, and thus allowing the conversion of work with software from manual to automatic. Also, in order to not damage the way legacy software work, and in order to abstain from replacing existing work modules for these software, the CalCul system provides a mirror between the changes done using the Python object and the legacy data files, and thus allows working dually on the software – either from CalCul or from the data files itself. CalCul also interfaces with IPython interactive command shell, and in this way, the user can manage all of his scientific actions entirely in one hermetic workspace.

Keywords. High-Performance Computing, Python Workspace, Legacy Computational Scientific Codes, Parallel Tools, Parameters-Sweep.

1. Introduction

Scientific computation is a central tool in computerized simulation planning, which runs on a powerful, parallel computer [1]. In order to perform their research jobs, physicists at NRCN use software developed by the computational groups (such as LEEOR2D-3D [2] [3] [4]), and other software such as OCTAVE [5] visualize and analyze results. In this paper, we will examine the workspace in which the physicist work while using the scientific software and the parallel computing, and provide a software system that will significantly improve the physicists' working processes.

In order to demonstrate the essence of the improvement, there is a need to understand first the workflow of a physicist in a computational group. Usually, the physicist has

performed a scientific job that includes running tens of computations, analyze their results, and produce numerous graphs that display the interesting results physically. After the physicist finishes the job, he also participated in a discussion where he was asked to perform the computations again with a change of one (or more) of the physical parameters and was also asked to test how it changes the physical conclusions that were previously presented. Then, the physicist returns to work and marks the computations he would like to recalculate. Afterwards, the physicist chooses which parameter he would like to change, inserts the new values and chooses which graphs he would like to generate from the new calculation results.

The experience in the computational field certainly demonstrates that this scenario is almost imaginary, or at the very least far from today's reality, since in order to make such a scenario occur in today's reality, one must access the folder where each calculation was made, make a copy of it, and edit the data file. Then, one should check which computers are available, and execute as many runs as possible. When these runs are terminated, more runs are executed, until all runs are terminated. Choosing the graphs we would like to generate means that after each run terminates, we need to access its folder from the visualizing software (in this case, OCTAVE), open the diagnostic files, extract the relevant data, and then integrate the results from numerous computations and create the graphs.

This description may sound familiar to anyone handling computations, and such operation may include additional struggles. If a part of the job has taken place numerous weeks or months ago, the physicist may not remember which computation is in each folder. In this case, the physicist must examine his folder tree, use the *diff* command to see the differences between the different folders, understand which computation exactly was created in each folder, and why. This process alone may take a few hours.

In the next stage, when the runs are being executed, they must be "supervised" to make sure they don't crash from I/O exceptions or other technical reasons, and if this happens, the runs must be restarted from the latest checkpoint. This way, even when the runs are being executed, they occupy the physicist's attention and disturb him from working on other subjects.

In the result analysis stage, the physicist goes through the original graphs he had generated, and generates equivalent graphs from the new runs. In the best case scenario, the graphs were generated using OCTAVE script files, which the physicist had stored properly, in which case they need to be located and executed again from the appropriate folder. In a less optimistic scenario, the graphs were generated interactively from the OCTAVE working window, which would mean the physicist would have to recall earlier work in order to regenerate equivalent graphs from the new calculations. Such task may also take quite some time.

The popular conception regarding this issue is that there is nothing to do about it and this is how things work: we copy folders, edit files, and run OCTAVE. It takes some time, but it is part of the job requirements of computational physicists. Additionally, another common belief is that no alternative solution is necessary since this is the way things have worked for many years and it works well: research is being done, and papers are being written. The authors of this paper would like to make two counter-arguments:

1. The working procedure described above will be insufficient in the future. It works well for one or a few computations. However, when tens of computations need to be executed, it can be problematic, and when hundreds or thousands computations are necessary, it is limiting and may be impossible. In the past, this procedure had not been a bottleneck, since the computation power had

limited the number of computations that could be performed. However, as the computation power increases (and it keeps increasing), the computer has the ability to perform more computations in a parallel fashion, which means the overhead of the entire process becomes more and more significant. To demonstrate, let us assume that today, a physicist spends an estimate of 10% of his time on technical tasks of executing and maintaining computations. Assume the next computer will have 20 times the computation power of the current computer. Assume we will use this increase of power to create more descriptive computations, but in parts of it, we will also test more parameters. Assume the number of computations that the physicist performs will increase fourfold. In today's existing methods, both the overhead time will increase fourfold, and we will reach a situation where 40% of the physicist's time is spent on technical tasks. We cannot accept such situations.

2. There is another way. The imaginary scenario from the opening paragraph is no longer unrealistic because, after all, all the information required for it to happen is already stored on the computer. The first computation has already been executed, and the corresponding graphs have been produced.
3. All of the additional information necessary for the computer is just which parameter needs to be changed, and which computations need to be repeated. To demonstrate this, let us assume every physicist hired an assistant that did not understand physics at all but can follow precise instructions. The physicist would tell him to take these and these folders, create their copies, change the following parameters in the copied folders, and run them. Then, execute all the OCTAVE commands in which he has seen the physicist execute on the original folders. It is clear that this human assistant would have all the information necessary to complete the task. Hence, a software system that will function just like this assistant is, in fact, possible. After all, the computer is the best in remembering how tasks had been executed in the past, and precisely follow instructions.

The first claim is obvious and does not require further discussion. The next sections will elaborate on the second argument and present general guidelines and initially describe the software system that will make us closer to the scenario described in the first paragraph. In section 2**,** the root of the gap for the current problems will be analysed against the ideal state described above. In section 3, we will discuss the Python programming language and explain why it is the appropriate tool to implement the suggested software system. In section 4 we will present the various components of the system and elaborate on the features of each component and its benefits. And finally, we will provide our conclusions.

2. Vocabulary Gap

The reason for encumbrance in the current working process is the gap of vocabulary between the physicist and the computer. The computer understands the following terms: folders, text files, computation servers, and executable files. These terms are closely related:

- A folder may contain files and other folders.
- A file or a folder is defined by its name and the folder containing it.

- A text file contains a collection of lines that contain a collection of characters.
- An executable file can be run on any computation server.
- When an executable file is run, it can read and write the content of files.

In addition to the rules that explain the relationship between these terms, the operating system includes a set of commands that allow the manipulation of the above objects:

- Commands that create folders and copy files.
- Commands that compare between text files based on the lines that make them up.
- Commands that allow running a particular executable file on particular computation servers.

This is the vocabulary of a computer, and it knows how to execute commands that are formatted using these terms. The physicist's world is different in that way: it includes computations, parameters, FORTRAN code, diagnostics, and graphs. These terms are also closely related:

- A computation is defined by a FORTRAN code with a collection of parameters.
- A computation may be in one of three states: currently being executed, has been executed, or has not been yet executed.
- When the computation is executed, it creates and writes diagnostics.
- From the diagnostics of numerous calculations, graphs can be generated.

In addition to the rules that relate these terms together, the physicist may want to engage with the above objects:

- Run a computation.
- Compare between computations and see the differences of the FORTRAN code and the set of parameters that define them.
- Generate a graph from a diagnostic created by some computation.
- Create a computation based on an existing computation and change some parameters.
- Generate a graph similar to a previously generated graph, based on a new calculation.
- Generate a graph based on the results of numerous calculations.

This is the physicist's world in this context, and he articulates his desires using these terms. Thus, we have a computer with a certain set of terms (which can accept commands that are formatted with these terms), while we have a physicist with a different set of terms (with desires formatted using these terms). In order for the computer to be able to perform the physicist's desires, these desired must be translated from using the same set of terms to another. The translation from the physicist language to the computer language is not complex:

- The FORTRAN file is located inside text files.
- Parameters are located inside a text file, called a data file.
- A computation is a folder that contains these folders.
- In order to compare two computations, the two text files need to be compared.
- In order to perform a computation, the compilation commands need to be executed from the computation's folder, and then they must choose the computation servers necessary, and perform the execution command.
- In order to test the execution's state, look at the last lines of a certain text file.
- In order to stop the execution, write a number to a certain text file.

- In order to examine the results of a computation, open the diagnostic files using OCTAVE.
- To generate a new computation based on an existing one, copy all files from one folder to a new folder and edit it.

In today's existing reality, the physicist himself performs the translation. This is the way physicists learn the computer language. They learn which commands copy files, which files compile, which command show available computers and which allow to compare text files and look at the last files of each file. The entire job of the physicist with a computer is done using a computer language that is far from the physicist's language, even though the translation is not complicated.

There is an alternative situation: while physicists learn the computer language, terms, and rules, we can teach the computer how to use the physicist terms and their rules. Such action is quite common in the field of software engineering; it is called translating from a language of problems to a language of solutions, or at least creating an additional abstraction layer [6]. Thus, is it possible to create a software system that will allow a physicist to work in a world of computations, parameters, diagnostics, and graphics, but only if we use the appropriate tools and methods.

3. Python – The Appropriate Tool

CalCul, the suggested system, is based on a Python environment and the common methodologies that are practiced in the environment. Python is a high-level, dynamic language that originated in 1991 [4]. In the years since, it had developed to become one of the most influential languages in the computer science world. Python is especially supporting and encouraging of fast and dynamic working methods of software development. In addition, many libraries have been developed – particularly scientific libraries – and due to the language's structure, all of these libraries interact with each other in synergy and enable the performance of complex task with an unprecedented ease. Python is used in many projects in the computation world in general and specifically in scientific computation [7].

Below is a summary list of the advantages of Python with regards to developing a working environment for physicists:

1. Dynamic: Python makes it very easy to create a prototype and quick development. In Python there is no need to analyse the complete needs of the system and then write the code; it is possible to quickly write the code for part of the needs, start using it, and later on add or change it according to the changes needed.
2. Modular: Python encourages and enables building modular software systems. This way, while every module in the system can function in isolation and provide some functionality, modules can work together and integrate to provide a richer functionality.
3. Mature tools for scientific computations: over the years, numerous modules have been developed in Python to provide almost all functionality that is offered by OCTAVE [7]. Additionally, advanced visualization tools such as VisIt [8] are integrated with a Python interface, along with other resource management systems for parallel computers [9] (such as SLURM).
4. Syntax: one of the leading principles that dominated the language design is the understanding the importance of readability. A Python code is usually more

readable and easier to maintain compared to code in other languages. Additionally, it is very easy to learn how to use Python, both for physicists and engineers. In this way, Python is like OCTAVE: it is easier to learn even for those who do not program professionally, but unlike OCTAVE, Python is a complete, strong programming language.

5. Programming fashion: Python supports both Programming in the Small and Programming in the Large - this means Python has proven support in both writing simple scripts that handle small tasks and in creating larger, infrastructure systems that consist of hundreds of thousands of lines of code [10].
6. Interactive work: Python supports both regular programming as well as equipping the programmer with strong tools for interactive tools, similar to the workspace of OCTAVE. Two relevant tools in this context are IPython [11], an improved Python environment for interactive jobs with clustering abilities, and Spyder [12], an OCTAVE-like, Python-based environment.

In the next section, we will describe the structure of the software system. The structure is not of a monolithic system, but of a modular one – which means a collection of individually independent modules. This way, it is possible at first to deliver the software using one module and get immediate benefit. Next, it is possible to implement additional models; the suggested models are ones that interact in a well-designed, proved manner, which means that as more modules are implemented, the utility of the modules together will be greater than the sum of utilities for each module independently.

4. The Computation Management Module

4.1. Current Calculations Management

A computation is defined by a set of files, usually FORTRAN files, and a data file. The commonly practiced procedure is that each computation is located in a different folder, and a folder that contains a terminated computation is not alternated anymore and remains constant, containing the result of the computation that was executed inside it.

When a physicist would like to create a new computation, he creates a new folder, copies the files from a similarly prepared computation, changes some parameters, and runs the computation.

Another aspect of the common working practices is that folders of a specific computation are often created as a subfolder of the folder of the computation on which it is based. This way, the folder tree structure also hints at the logical structure of the computation – a computation that is under a specific folder is a variation of the computation in the folder above it. For example, a folder tree may look like this:

```
>> tree("project_A")
project_A/
-- 3d/
     |-- param_a=3.1/
     |   |-- different_rezone/
     |   |   -- high_res/
     |   -- high_res/
     -- param_a=3.2/
```

In this example, the folders *project_A/3d* contains a three-dimensional computation for a certain project, while the folders *params_a=3.1* and *params_a=3.2* contain the same computation with the difference of some parameter with different values. Inside the folder *params_a=3.1*, the tests of a different rezones are located which contains a higher resolution.

To demonstrate this, let us examine the process of reviewing a computation that has been performed in the past, and create a new computation with a higher resolution. Today, the process looks like this:

```
>> pwd
/physicist/project_A/3d/param_a=1
>> mkdir high_res
>> cp ./* high_res
>> cd high_res
>> pwd
/physicist/project_A/3d/param_a=1/high_res
>> vi data_file # Edit the data file and increase resolution
>> make clean & make & make run
>> cat out | grep "NCYC=" | tail
```

This is the process of a single computation. Now, assume that the original folder */physicist/project_A/3d/param_a=1* has 10 more computations with different values of *param_a*, and that the physicist is interested in making computations with higher resolution for all values. In this case, the process above will be executed 10 times manually. As previously explained, this work fashion should not be tolerated.

4.2. CalCul Calculations Management

The main module in the CalCul system is the computation management module. This module is responsible to the computations in every phase they go through: running computations, terminated computations, and computations that are being prepared. The purpose of this module is to create the entire process of creating and managing calculations simpler and easier. This is achievable by changing the process from a user controlled process that is based on the file system, to a process controlled by an automatic, customized software system that handles computation management. Also, in order not to damage the way legacy software work [13], and in order to abstain from replacing existing work modules for these software, the CalCul system provides a constant mirroring between the changes done using the Python object and the legacy data files, and thus allows working dually on the software – either from CalCul or from the data files itself. This has been done using the *datafile* method, which is able to automatically convert standard legacy codes data files (i.e. textual files of numbers, strings and structs) to a Python object, which can be operated using dedicated method. Among those methods we can mention:

Table 1. CalCul main methods.

Command	Info
clone	*Copy files / directories to other destination.*

archive	*Prints calculation under directory including selection base on keyword and order by time or hierarchy.*
tree	*Prints directories under path as a tree, options to pad and select directories or files presentation.*
tocopy	*Returns a deep copy of a class instance. Keeps each copy in a specified directory under directory /object_copies in wd.*
put	*Creates subdirectories in current or other directory with a data file object. The path to the subdirectory with the datafile object will start with "calc", and can get a list with the desired name of the sub-directory base on different arguments.*
runc	*Compile and run the calculations using SLURM*
status	*Get status of the runs by SLURM jobid*
datafile	*Constructor - Gets full path to a data file or operates from a path in which a data file exists*
plot	*Prints the datafile object by categories and data-types using specialized identifying colors. Can print the whole file or specific range of lines.*
replace	*Replace an argument in the datafile object as well as in the datafile file itself. Can replace any argument in the datafile by indexes ([line, arg]). Can get an expression or formula, and a second argument with the specific values, and the outcome will be calculated automatically and will be set in the object, as well in the file itself. All changes done using other functions use replace.*
repll / repla	*Replace line by a specific index with a different line / Replace argument by an argument name index*
find	*Finds all of the lines in which the expression searched found.*
get	*Get the value from the file in its type.*
get_type	*Returns the value and type information from the file in its type.*
getl / addl / reml	*Get/Add/Remove line by a specific index.*
addarg / remarg	*Add / Remove argument by a specific index in a line.*
undo / redo	*Returns the datafile object and the file itself to the previous / next state (for replacements).*

So far, we have seen how the computation management module can support every work process of today's physicists. However, we have not yet seen which new features it enables. First, it is important to note that the first criteria which the module needs to support are that any ability that is easy to perform for physicists today, will be easy to perform on the new module as well. However, the computation management model allows doing things that today are difficult or impossible. As previously described, assume that the original folder */physicist/project_A/3d/param_a=1* has 10 more computations with different values of *param_a*, and that the physicist is interested in

making computations with higher resolution for all values. Using the computation management module CalCul, the same can look like this, when executed from Python's interactive workspace:

```
In [1]: old_calcs = archive('/physicist/project_A/3d')
In [2]: old_calcs
Out[3]: ['/physicist/project_A/3d/param_a=1/data_file',
         ...
         '/physicist/project_A/3d/param_a=10/data_file']
In [4]: clone('/physicist/project_A/3d/*',
                                '/physicist/project_A/3d/high_resolution')
In [5]: new_calcs = archive('/physicist/project_A/3d/high_resolution')
In [6]: new_calcs
Out [6]: ['/physicist/project_A/3d/high_resolution/param_a=1/data_file',
         ...
         '/physicist/project_A/3d/high_resolution/param_a=10/data_file']
In [7]: new_res = 100
   ...: for new_calc in new_calcs:
   ...:     new_calc.repla('N_CELLS', 1, new_res)
In [8]: for c in new_calcs:
   ...:     print runc(c)
Out [8]: {Outs and Errors of the Run}
In [9]: for c in new_calcs:
   ...:     print status(c)
Out [9]: JOBID 5034985 -> Slurm=No Errors | Running=None | Normal-
Termination=True
```

The most important observation in this code is that a computation is represented by a Python object. The computation object has functions that it provides (methods), and the computations archive has functions that return a computation object or a list of computation objects. We see that at first, we called the function *archive;* this function returns a list of *Paths* that represent the chosen computations (which can be accessed as object using the *datafile* function). Then, we wrote a short loop that iterates over the chosen computations, and generates new computations from them. After examining the new computations, an additional loop executes them and they start running. An additional loop monitors their performance until termination.

Aside from choosing the computations, there are two non-trivial actions performed in the above example: changing the resolution in the data file (*data_file)*, and running the computation. The change of resolution was done using textual replacement of a string inside the computation's data file. In order to provide such an option, all the computation object needs to provide is a convenient access to the *data_file* file that defines the computation. In the above example, this access is done using the function *repla.*

As for executing the computation, in the above example, the computation object provides a function called *runc* that runs it. The implementation of this function accesses the computation folder and executes the running commands that will execute it in the currently connected computation server using the SLURM resource management system [14]. It should be noted that SLURM contains a convenient Python interface (called *PySlurm),* which allows to easily implement that connection between the computation management model and the resource management system for various uses.

Summary

In recent years, there has been a significant increase in the computational power, simultaneously to the amount of computations performed by a physicist. However, due to the fact that simulation software tools are mostly legacy software, it was impossible to convert the working environment with these tools to modern tools. This paper presents a software system called CalCul. The purpose of the presented system is to allow physicists to improve work efficiency while using a module that automatically decodes the data files of programs and converts them, along with the computational array they represent, to a Python object, and thus allowing the conversion of work with software from manual to automatic.

References

[1] Eijkhout, Victor. *Introduction to High Per' Scientific Computing*. Lulu. Com, 2014.
[2] Freed N, Ofer D, Shvarts D, Orszag SA. *Two-phase flow analysis of self-similar turbulent mixing by Rayleigh–Taylor instability*. Physics of Fluids A: Fluid Dynamics (1989-1993). 1991 May 1;3(5):912-8.
[3] Ofer D, Alon U, Shvarts D, McCrory RL, Verdon CP. *Modal model for the nonlinear multimode Rayleigh–Taylor instability*. Physics of Plasmas (1994-present). 1996 Aug 1;3(8):3073-90.
[4] Malamud G, Levi-Hevroni D, Levy A. *Two-dimensional model for simulating shock-wave interaction with rigid porous materials*. AIAA. 2003 Apr;41(4):663-73.
[5] Brewster, Matthew, and Matthias K. Gobbert. *A comparative evaluation of Matlab, Octave, FreeMat, and Scilab on tara.* Citeseer2011 (2011).
[6] Ghezzi, Carlo, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering*. Prentice Hall PTR, 2002.
[7] Oliphant, Travis E. *Python for scientific computing*. Computing in Science & Engineering 9.3 (2007).
[8] Harrison, Cyrus, and Hari Krishnan. *Python's Role in VisIt.* Proceedings of the eleventh annual Scientific Computing with Python Conference (SciPy 2012). 2012.
[9] Thiell, Stéphane, Aurélien Degrémont, and Henri Doreau. *ClusterShell, python library and tools for scalable cluster administration.* PyHPC. 2013.
[10] Lutz, Mark. *Learning Python: Powerful Object-Oriented Programming*. O'Reilly Media, Inc., 2013.
[11] Pérez, Fernando, and Brian E. Granger. *IPython: a system for interactive scientific computing.* Computing in Science & Engineering 9.3 (2007).
[12] Boschetti, Alberto, and Luca Massaron. *Python data science essentials*. Packt Publishing Ltd, 2016.
[13] Feathers, Michael. *Working effectively with legacy code*. Prentice Hall Pro', 2004.
[14] Yoo, Andy B., Morris A. Jette, and Mark Grondona. *Slurm: Simple linux utility for resource management.* Workshop on Job Scheduling Strategies for Parallel Processing. Springer, Berlin, Heidelberg, 2003.

Acknowledgments

This work was supported by the Lynn and William Frankel Center for Computer Science.