INDIAN JOURNAL OF SCIENCE AND TECHNOLOGY

RESEARCH ARTICLE

OPEN ACCESS

Received: 25-03-2024

Accepted: 28-05-2024

Published: 12-06-2024

Citation: Arora I (2024) Improving Performance of Data Science Applications in Python. Indian Journal of Science and Technology 17(24): 2499-2507. https://doi.org/10.17485/IJST/v17i24.914

* **Corresponding author**.

indu.arora@mcmdavcwchd.in, indarora@yahoo.co.in

Funding: None

Competing Interests: None

ISSN
Print: 0974-6846
Electronic: 0974-5645

Improving Performance of Data Science Applications in Python

Indu Arora[1]*

1 Associate Professor, Mehr Chand Mahajan DAV College for Women, Chandigarh, India

Abstract

Objectives: The objective of this paper is to focus on the need of performance optimization of Data Science applications with an aim to enhance the speed, responsiveness, and effective resource utilization. **Methods:** This paper identifies the factors which effect the performance of an application and then classifies them into internal and external factors. This study mainly focuses on internal factors such as execution speed and space requirement. It uses code developed in Python as use cases to explore and examine features which improve the performance and effectiveness of Data Science (DS) applications. **Findings:** This paper provides comprehensive way of improving performance of applications by implementing features of Python like generator, vectorization, profiling, concurrency & parallelism, and caching. It is observed that space requirement reduces with the use of generators in code, leading to better space management. The conventional way of writing code in Python required additional 5.2 MiB (Mebibyte) space, while there was no additional space requirement with the use of generators in the same code. It is further observed that vectorization feature of Python helps in reducing execution time in comparison to the code written without vectorization. This paper further uses parallelism in the code and gets approximately 68% improvement in execution time. The implementation of profiling feature helps in identifying the time and space requirement which can be improved using features like generator, vectorization, concurrency, and parallelism. Other features like handling I/O operations, use of caching and choice between flat file data and database are also deliberated to improve the performance of code. **Novelty:** This paper highlights the potential of improving performance of DS applications using generator, vectorization, concurrency and parallelism features of Python and examines the effectiveness of the claim through its implementation.

Keywords: Data Science; Generator; Vectorization; Parallelism; Profiling

1 Introduction

Performance of any software application, may be web, desktop or mobile application, is measured on its ability to respond to a query made. Further, these applications can be

Online Transactional Processing (OLTP) or Online Analytical Processing (OLAP). The purpose of OLTP applications is to process transactional data, while the OLAP is to analyze existing data. Most of the e-commerce web sites heavily rely on OLTP applications[1]. The tasks of OLTP applications include insert, delete, query and update operations. On the other hand, the purpose of OLAP is to discover trends by aggregating data, applying complex analytical estimations, and providing different report views[2]. OLAP works with data retrieved from multiple sources and data warehouses. An OLAP application does the job of querying and presenting data. DS applications, Business Intelligence Applications and Decision Support Systems use OLAP. The focus of DS applications is to extract useful information from structured and/or unstructured data and forecasts trends.[3] DS is a multi-disciplinary field which comprises set of principles, algorithms, and processes to extract useful information and patterns from large volume of data that help business organizations to make business decisions[4]. It uses data models, data analytics, computations and learning to convert data to information and information to knowledge and then to decision-making. However, a DS application is expected to provide useful information accurately and timely after processing the data irrespective of whether it is OLTP and OLAP kind of applications. So, the performance of an application is a critical non-functional requirement.

Performance Optimization is the process to improve any software application so that the application can be executed efficiently and rapidly while maintaining the results of data analyses accurate. Performance can be optimized by focusing on both front-end and back-end components of the software application or on internal factors like time and execution, space management and I/O operations and external factors like configuration of hardware, network bandwidth and latency and the characteristics of data. Though external factors like hardware, memory model and cache designs have a significant impact on the performance of a software, it is often beyond the direct control of the development team[5]. Internal factors affect the performance of a software but can be managed through software itself. Application developers can evaluate algorithmic choices, eliminate redundant computations, implement algorithms with better time and space complexity, implement efficient memory allocation and deallocation strategies, optimize database schema and queries, implement caching effectively, and explore and identify parallelization opportunities for the optimization of code through software. The ultimate aim of optimizing an application is to reduce execution time, memory, space usages and the seek time of data fetching for improving performance. The application can be monitored and analyzed to improve its performance on these parameters[6]. There is always a trade-off between these parameters, for example, an increase in the memory requirement can reduce execution time required for the application or vice versa.

This research paper deliberates the use of Python, a forerunner language used for DS applications and comprehensively explores the impact of Python features like generator, vectorization, concurrency, and parallelism on internal factors.

2 Methodology

Python, a high-level programming language, is being used in many data science and machine learning related applications. Python provides comprehensive libraries, built-in data structures, dynamic type system and memory management features. Its interpreters are available for many Operating Systems[7]. Its rich features can be used for improving performance of software by application developers[8]. This paper adopts the following methodology.

- The features like generators, vectorization, parallelism, and caching are identified and implemented in DS applications developed in Python to examine the improvement in performance.
- Sample dataset of student performance is taken from popular website kaggle.com. In addition to this dataset, random data is also used.
- The impact of identified features is analyzed by comparing code written using conventional method with the code incorporating special features of generator, vectorization, concurrency, and parallelism.
- Both types of code are executed on same data for comparing results in terms of execution time and space used.
- The benefits of other features like I/O operations and caching are also deliberated that can enhance the performance of an application.

3 Result and Discussion

3.1 Use of Generators

Generator allows to iterate over large datasets without creating entire sequence in memory. The use of Generators improves the memory usage especially working with large datasets. It provides one value at a time without storing entire intermediate values in memory[9]. The main benefits of using Generator are minimal memory consumption, lazy evaluation and simpler syntax. The purpose of lazy evaluation is to defer evaluation of expression until its value is needed and this way, repeated evaluation is avoided. Memory usage becomes more efficient when a file is read line by line using Generator and it is better than loading entire file into memory. Python provides a function called Generator that uses yield keyword instead of return keyword to produce series of values. In order to prove that Generator is more memory efficient, a Fibonacci series of 10000 elements is created in Python using both conventional method and the Generator. Then, memory usage of both the codes is analyzed.

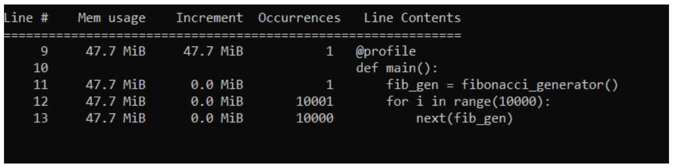
The following lines show the Python code using conventional method.

```
import cProfile
from memory_profiler import profile
#run using python -m memory_profiler <filename.py>
def fibonacci_generator(n):
   fibonacci_series = [0, 1]
   while len(fibonacci_series) < n:
      next_number = fibonacci_series[-1] + fibonacci_series[-2]
      fibonacci_series.append(next_number)
   return fibonacci_series
@profile
def main():
   ret=fibonacci_generator(10000)
if __name__ == '__main__':
   main()
```

The following lines show the Python code using Generator.

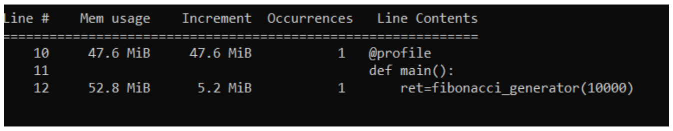
```
import cProfile
from memory_profiler import profile
#run using python -m memory_profiler <filename.py>
def fibonacci_generator():
   a, b = 0, 1
   while True:
      yield a
      a, b = b, a + b
@profile
def main():
   fib_gen = fibonacci_generator()
   for i in range(10000):
      next(fib_gen)
if __name__ == '__main__':
   main()
```

Figures 1 and 2 show the memory usage using Generator and Conventional methods respectively.



```
Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
     9     47.7 MiB     47.7 MiB           1   @profile
    10                                         def main():
    11     47.7 MiB      0.0 MiB           1       fib_gen = fibonacci_generator()
    12     47.7 MiB      0.0 MiB       10001       for i in range(10000):
    13     47.7 MiB      0.0 MiB       10000           next(fib_gen)
```

Fig 1. Use of Generator



```
Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    10     47.6 MiB     47.6 MiB           1   @profile
    11                                         def main():
    12     52.8 MiB      5.2 MiB           1       ret=fibonacci_generator(10000)
```

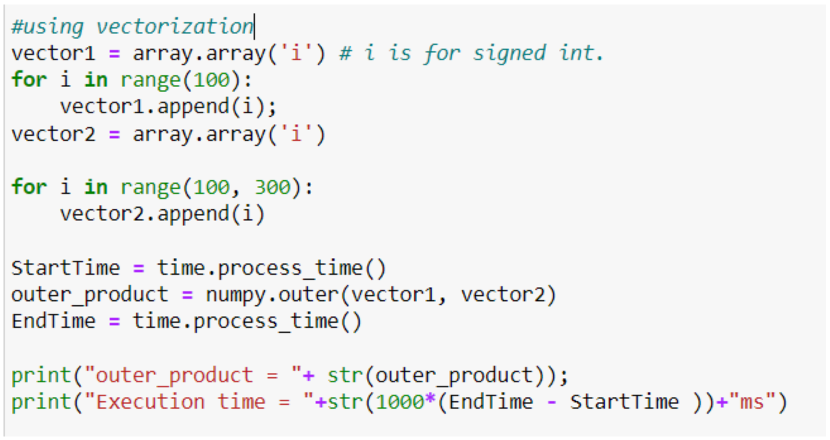
Fig 2. Conventional Method

The result shows that when a conventional method is used, it takes 5.2 MiB (Mebibyte) additional memory as compared to 0.0 MiB when Generator is used where 1 MiB is equal to 1.048576 MB. This result may differ slightly when it is executed on different machines.

3.2 Use of Vectorization

Vectorization is a powerful technique that is used to speed up operations involving scientific computations and data manipulations and to make computation more efficient. It provides standard mathematical functions to be applied on entire data array or matrix at once without using explicit loops. It improves the time required to perform the operations. Applying loop over an array is a slow process in Python. Standard mathematical functions are preferred in Python to perform operations faster on an entire array rather than using loops. One such library is NumPy. The Outer, dot and multiply functions are used for Vectorization. In order to demonstrate the efficiency of outer product using vectorization over classical method, a Python program is written to perform outer product of two vectors, n x 1 and 1 x m which results in a matrix of n x m. Execution time of both the codes is evaluated on same set of vectors. The screenshots of partial codes for both methods are given in Figures 3 and 4 respectively.

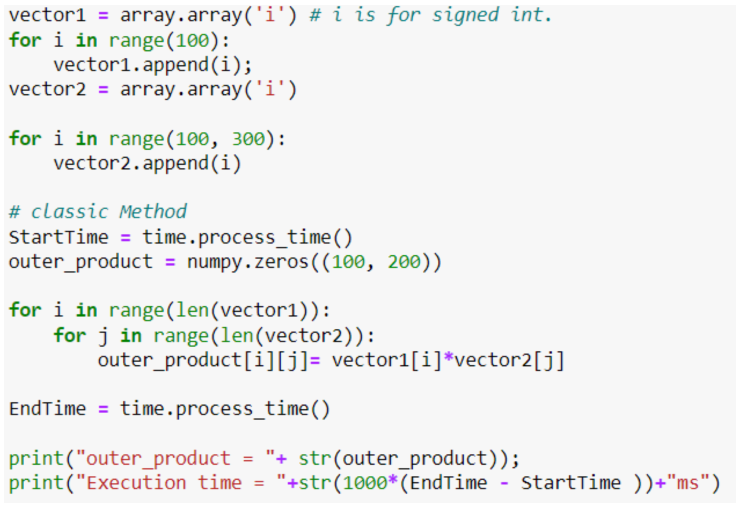
Using Vectorization Technique



```
#using vectorization
vector1 = array.array('i') # i is for signed int.
for i in range(100):
    vector1.append(i);
vector2 = array.array('i')

for i in range(100, 300):
    vector2.append(i)

StartTime = time.process_time()
outer_product = numpy.outer(vector1, vector2)
EndTime = time.process_time()

print("outer_product = "+ str(outer_product));
print("Execution time = "+str(1000*(EndTime - StartTime ))+"ms")
```

Fig 3.

Using Classical Method (with loops)



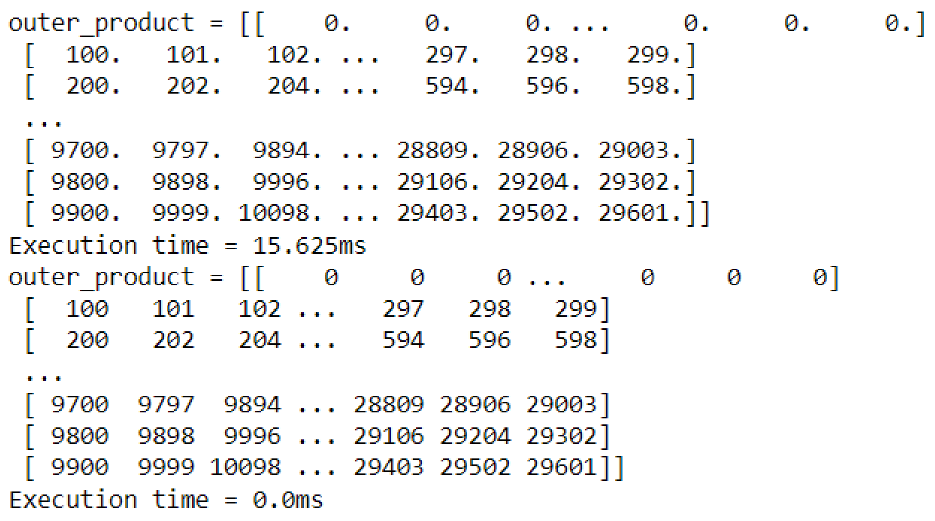
```
vector1 = array.array('i') # i is for signed int.
for i in range(100):
    vector1.append(i);
vector2 = array.array('i')

for i in range(100, 300):
    vector2.append(i)

# classic Method
StartTime = time.process_time()
outer_product = numpy.zeros((100, 200))

for i in range(len(vector1)):
    for j in range(len(vector2)):
        outer_product[i][j]= vector1[i]*vector2[j]

EndTime = time.process_time()

print("outer_product = "+ str(outer_product));
print("Execution time = "+str(1000*(EndTime - StartTime ))+"ms")
```

Fig 4.

The output of codes of both the methods is given in Figure 5.

Mathematically, the outer operator with vector1 of 3x1 and vector2 of 1x3 dimensions is illustrated in Figure 6.

Similarly, execution time of other operations using vectorization is found lesser in comparison of classical methods.



```
outer_product = [[    0.     0.     0. ...     0.     0.     0.]
 [  100.   101.   102. ...   297.   298.   299.]
 [  200.   202.   204. ...   594.   596.   598.]
 ...
 [ 9700.  9797.  9894. ... 28809. 28906. 29003.]
 [ 9800.  9898.  9996. ... 29106. 29204. 29302.]
 [ 9900.  9999. 10098. ... 29403. 29502. 29601.]]
Execution time = 15.625ms
outer_product = [[    0     0     0 ...     0     0     0]
 [  100   101   102 ...   297   298   299]
 [  200   202   204 ...   594   596   598]
 ...
 [ 9700  9797  9894 ... 28809 28906 29003]
 [ 9800  9898  9996 ... 29106 29204 29302]
 [ 9900  9999 10098 ... 29403 29502 29601]]
Execution time = 0.0ms
```

Fig 5.

$$\text{Vector 1} = \begin{bmatrix}1\\2\\3\end{bmatrix} \quad \text{Vector2} = [1\ 2\ 3\ ], \text{then Outer Product} = \begin{bmatrix}1 & 2 & 3\\2 & 4 & 6\\3 & 6 & 9\end{bmatrix}$$

Where dimension of Vector1 is 3x1 and that of Vector2 is 1x3. The dimension of result outer product is 3x3.

Fig 6.

3.3 Use of Profiling Tools

The purpose of Profiling tools is to identify the bottlenecks in terms of time and space, if optimization is a major concern[10]. Python provides several Profiling libraries like cProfile, Profile, Line_profiler, memory_profiler and Py-Spy among others. The library cProfile provides the information of execution time of a function and number of times a function is called. This helps in analyzing the execution time. After analyzing the code, bottlenecks are identified and then code is re-written. It is used in code already given in Section 3.1 of this research paper.

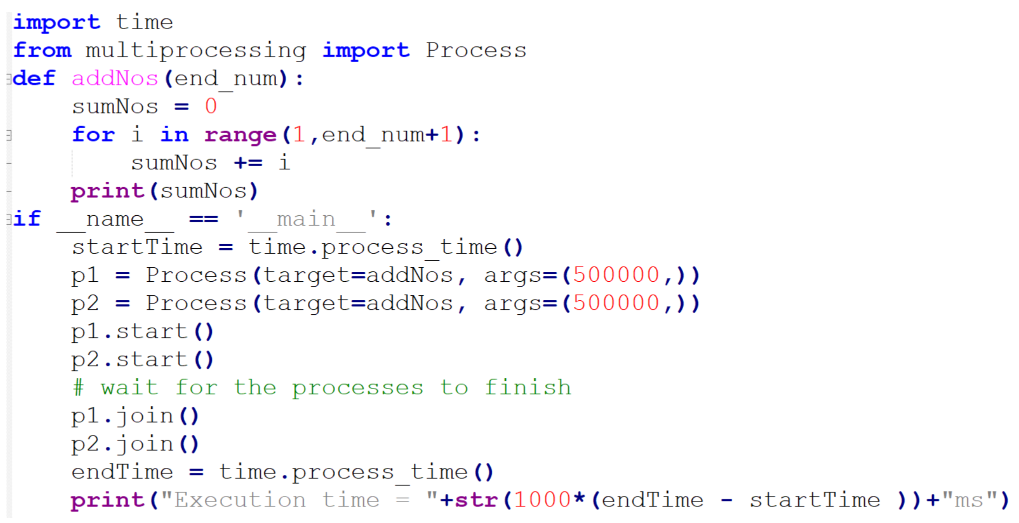
3.4 Use of Concurrency and Parallelism

Python also employs concepts of concurrency and parallelism for improving performance and efficiency of code. Concurrency is used to handle mutual exclusive tasks concurrently and is achieved using multitasking and multithreading while Parallelism is a process of executing multiple tasks simultaneously using different resources such as CPU in distributed environment. A task is divided into subtasks and each subtask is executed independently on different hardware resources. Concurrency and parallelism in Python can be achieved through threading, multiprocessing, concurrent.features and asyncio techniques. In multithreading, multiple threads within single process are executed concurrently. In multiprocessing, threads are executed in parallel. Under these techniques, each thread uses a different CPU and other resources. Concurrent.feature is used to execute tasks using both thread based and process based techniques. Asyncio is specially used for I/O bound operations. To experiment with parallelism, a python code to add first 5,00,000 numbers is written using both parallelism technique and conventional execution technique. The partial code is given in Figure 7.

With Parallelism Technique

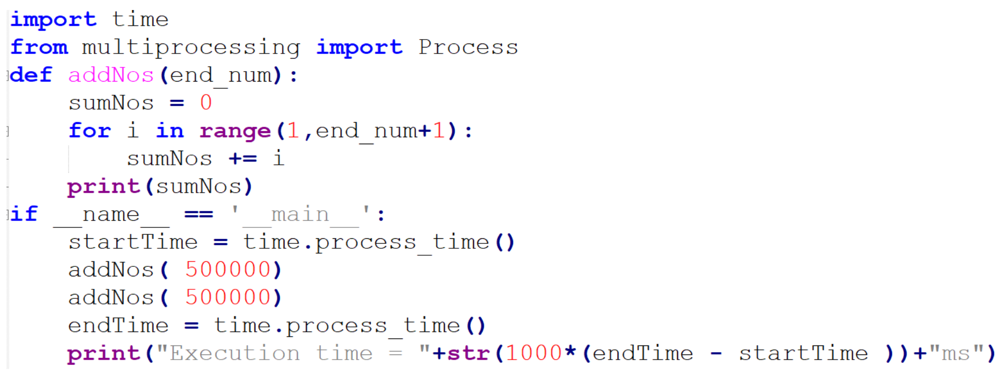
With conventional Execution Technique

Each program was executed five times to record the execution time which is summarized in Table 1. It is observed that average time taken to perform above task using parallelism technique is better than that of conventional execution method.



```
import time
from multiprocessing import Process
def addNos(end_num):
    sumNos = 0
    for i in range(1,end_num+1):
        sumNos += i
    print(sumNos)
if __name__ == '__main__':
    startTime = time.process_time()
    p1 = Process(target=addNos, args=(500000,))
    p2 = Process(target=addNos, args=(500000,))
    p1.start()
    p2.start()
    # wait for the processes to finish
    p1.join()
    p2.join()
    endTime = time.process_time()
    print("Execution time = "+str(1000*(endTime - startTime ))+"ms")
```

Fig 7.



```
import time
from multiprocessing import Process
def addNos(end_num):
    sumNos = 0
    for i in range(1,end_num+1):
        sumNos += i
    print(sumNos)
if __name__ == '__main__':
    startTime = time.process_time()
    addNos( 500000)
    addNos( 500000)
    endTime = time.process_time()
    print("Execution time = "+str(1000*(endTime - startTime ))+"ms")
```

Fig 8.

Table 1. Execution Time

Round	Execution Time in ms (With Parallelism)	Execution Time in ms (With Normal Execution)
1	15.625	46.875
2	0.000	62.500
3	15.625	62.500
4	15.625	31.250
5	31.250	46.875
Average	15.625	50.000

3.5 Use of I/O Operation

Another practice to improve performance of Python code is to optimize I/O (Input/Output) operations which are considered major bottlenecks when large data is fetched or file operations like reading or writing are done. Approaches like Buffered I/O, Batch I/O, Context Manager, Appropriate File Modes, Memory Mapping etc. are followed to improve I/O Operations. Buffered I/O deals with byte streaming and is used generally for non-text data. Batch I/O takes advantage of reading/writing data with larger batch size. Context Manager uses 'with' statement to work with files or resources that are required to be closed once operation is over. This technique ensures that resources are closed properly. Appropriate file mode also plays important role in improving performance, for example, use of 'rb' or 'wb' modes is preferred to 'r' or 'w' modes while working with binary files. The purpose of Memory mapping is to map portion of memory to a disk file. Programs directly read data from memory in case of memory mapping approach instead of using any I/O routine.

3.6 Use of Caching

The purpose of caching is to store frequently used data or results of some calculations for later use. It is one of the major optimizing techniques used especially in machine learning applications which require large set of data. The prefetching data in cache will reduce data access time. Caching techniques are categorised into two patterns, Spatial Caching and Temporal Caching. Spatial Caching is used in technical applications where data size is large. Data is kept in cache before it is used again and again. Temporal Caching is widely used to retain data in cache based on frequency. Commonly used caching techniques are FIFO (First In First Out), LIFO (Last In Last Out) and LRU (Least Recently Used).

3.7 Use of Database or Flat Files

Performance of code also depends on the source and format of data and whether it is from a database or from a flat file. Data Science applications heavily rely on large set of data therefore its format does matter. In flat files, data is stored in plain text while in database, data is stored in a particular format. Data is read sequentially from a flat file while direct access is possible in a database. Since, a flat file has size limitation, therefore, databases are used when data size is too large. Moreover, SQL queries makes it easy to manage data in databases. Therefore, choice between a database and a flat file depends on various factors. Few factors are listed in Table 2.

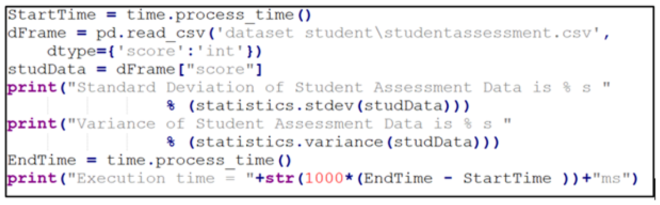
Table 2. Difference between a Flat File and a Database

Sr. No.	Factor	Flat File	Database
1	Scalability	Limited	High
2	Structure of data	Plain text	Formatted
3	Accessibility	Sequential	Direct through SQL (more efficient)
4	Size	Limited	Large
5	Security	Poor	High
6	Organization	Single/multiple files without any relationship	Relationship exists among different tables

Improving Performance of a Data Science Application

The factors or features explored in previous sections, can be used to improve performance of a Data Science application. For this, a sample dataset of student performance, available at https:// www.kaggle.com/datasets/rocki37/open-university-learning-analytics-dataset/ is downloaded[11]. The dataset contains multiple csv files. For testing purpose, one csv file, named as studentassessment.csv, is taken. The file has five fields, id_assessment, id_student, date_submitted, is_banked and score. It has 1,73,912 records. This data is used to calculate Variance and Standard Deviation (SD). Standard Deviation is the measure of how dispersed the data is in relation to the mean of the data. Low SD means data is tightly clustered around the mean while higher SD means data is scattered far away from its mean. Variance, on the other hand, measures average degree to which each point differs from its mean. SD is square root of variance.

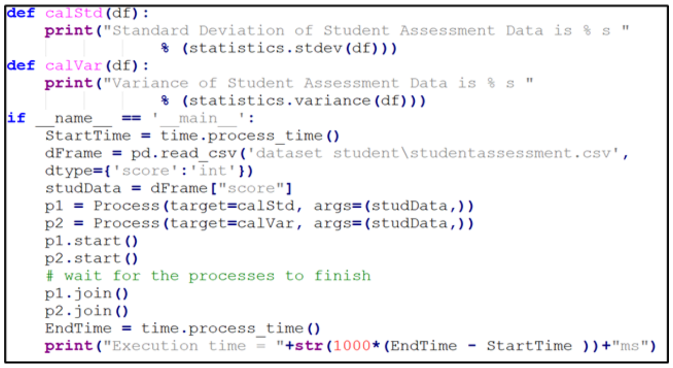
Three different ways are used to calculate Variance and SD in Python code. First method calculates Variance and SD using statistical library, second uses parallelism concept and the third method calculates Variance first using statistical library and then its square root i.e., SD is calculated using Math library. Figures 9, 10 and 11 show the partial implementation in Python. Execution time taken by each implementation is recorded five times and is given in Table 3.



```
StartTime = time.process_time()
dFrame = pd.read_csv('dataset student\studentassessment.csv',
    dtype={'score':'int'})
studData = dFrame["score"]
print("Standard Deviation of Student Assessment Data is % s "
                % (statistics.stdev(studData)))
print("Variance of Student Assessment Data is % s "
                % (statistics.variance(studData)))
EndTime = time.process_time()
print("Execution time = "+str(1000*(EndTime - StartTime ))+"ms")
```

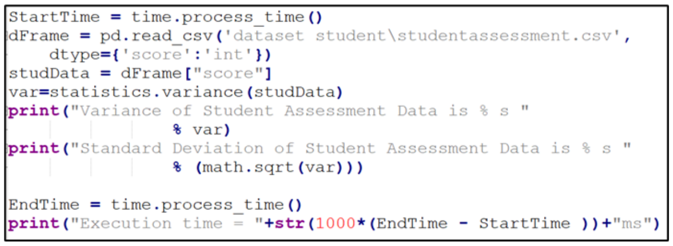
Fig 9. Partial code to calculate SD and Variance using Statistical Library

From Table 3, it is clear that execution time taken by program using parallelism is better than both the other cases. It also shows that if related measures (variance and SD) are to be calculated, then it is better to do them in a sequence. The case where variance is calculated first and then from variance, standard deviation is calculated, provides better execution time than to calculate both individually.



```
def calStd(df):
    print("Standard Deviation of Student Assessment Data is % s "
            % (statistics.stdev(df)))
def calVar(df):
    print("Variance of Student Assessment Data is % s "
            % (statistics.variance(df)))
if __name__ == '__main__':
    StartTime = time.process_time()
    dFrame = pd.read_csv('dataset student\studentassessment.csv',
    dtype={'score':'int'})
    studData = dFrame["score"]
    p1 = Process(target=calStd, args=(studData,))
    p2 = Process(target=calVar, args=(studData,))
    p1.start()
    p2.start()
    # wait for the processes to finish
    p1.join()
    p2.join()
    EndTime = time.process_time()
    print("Execution time = "+str(1000*(EndTime - StartTime ))+"ms")
```

Fig 10. Partial code to calculate SD and Variance using Parallelism



```
StartTime = time.process_time()
dFrame = pd.read_csv('dataset student\studentassessment.csv',
    dtype={'score':'int'})
studData = dFrame["score"]
var=statistics.variance(studData)
print("Variance of Student Assessment Data is % s "
                % var)
print("Standard Deviation of Student Assessment Data is % s "
                % (math.sqrt(var)))

EndTime = time.process_time()
print("Execution time = "+str(1000*(EndTime - StartTime ))+"ms")
```

Fig 11. Partial code to calculate Variance first and then SD

Table 3. Record of Execution Time

Round	With Normal Procedure	With Parallelism	Variance and then SD
1	187.50	46.87	125.00
2	187.50	62.50	125.00
3	171.87	62.50	156.25
4	171.87	93.70	125.00
5	156.25	78.12	125.00
Average	175.00	68.74	131.25

4 Conclusion

Performance of any software application in Python, especially a Data Science application, depends on many factors like use of generator, vectorization, profiling, concurrency & parallelism, I/O operations, caching and sequencing of operations etc. There is always a trade-off between optimum memory usage and execution time depending on the requirements. It was found that 5.2 MiB additional space was required when conventional approach of programming was used. With the use of generators on same data, no additional space was required. Similarly, findings in this paper endorsed that execution time was less using vectorization in comparison to conventional code. Parallelism approach further showed the improvement in execution time by 68% against the normal execution of the program. Profiling feature helped in identifying bottlenecks in the source code. This feature has been found useful in improving the performance. Other features like handling I/O operations, use of caching and choice between flat file data and database were also found useful in improving performance. Since external factors like hardware also plays an important role in the performance of an application. This factor can be explored in the future. The future scope also includes the study of other factors that are required to optimize Data Science applications.

References

1) El-Sayed N, Sun Z, Sun K, Mayerhofer R. OLTP in Real Life: A Large-scale Study of Database Behavior in Modern Online Retail. In: 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS);vol. 1. IEEE. 2021. Available from: https://doi.org/10.1109/MASCOTS53633.2021.9614295.

2) Vatresia A, Johar A, Utama FP, Iryani S. Automated Data Integration of Biodiversity with OLAP and OLTP. *Journal of Information System (e-journal)*. 2020;7(2):80–89. Available from: https://journal.unika.ac.id/index.php/sisforma/article/view/2817/pdf.
3) Chinthamu N, Karukuri M. Data Science and Applications. *Journal of Data Science and Intelligent Systems*. 2023;1(2):83–91. Available from: https://doi.org/10.47852/bonviewJDSIS3202837.
4) Schaad P, Ben-Nun T, Hoefler T. Boosting Performance Optimization with Interactive Data Movement Visualization. In: Proceedings of SC22: International Conference for High Performance Computing, Networking. 2022;p. 1–16. Available from: https://dl.acm.org/doi/10.5555/3571885.3571970.
5) Kandula YS, Madireddy BB, Sourab BR. Analysis of Hardware Impact on Software Performance. *International Research Journal of Engineering and Technology*. 2021;8(12):1326–1332. Available from: https://www.irjet.net/archives/V8/i12/IRJET-V8I12223.pdf.
6) Katsaragakis M, Papadopoulos L, Konijnenburg M, Catthoor F, Soudris D. A memory footprint optimization framework for Python applications targeting edge devices. *Journal of Systems Architecture*. 2023;142. Available from: https://doi.org/10.1016/j.sysarc.2023.102936.
7) Podoba V. 25 Tips for Optimizing Python Performance. 2023. Available from: https://www.softformance.com/blog/how-to-speed-up-python-code/.
8) Castro O, Bruneau P, Sottet JS, Torregrossa D. Landscape of High-Performance Python to Develop Data Science and Machine Learning Applications. *ACM Computing Survey*. 2023;56(3):1–30. Available from: https://doi.org/10.1145/3617588.
9) Prabagar S, Vinay K, Nassa, Senthil VM, Abhang S, Pravin P, et al. Python-based social science applications' profiling and optimization on HPC systems using task and data parallelism. *The Scientific Temper*. 2023;14(3):870–876. Available from: https://doi.org/10.58414/SCIENTIFICTEMPER.2023.14.3.48.
10) Ondiwa M. Python Object Caching - How does Python Optimize Memory Management for Integers?. 2023. Available from: https://micahondiwa.hashnode.dev/python-object-caching-how-does-python-optimize-memory-management-for-integers.
11) Russel R. Open University Learning Analytics Dataset: Course, Student and Assessment Data. . Available from: https://www.kaggle.com/datasets/rocki37/open-university-learning-analytics-dataset/.