Accelerating Python Code with Parallel I/O

Robin Varghese[1], Hashirul Quadir[1], Ladjel Bellatreche[2], and Carlos Ordonez[1]

1=Department of Computer Science, University of Houston, USA
2=LIAS/ISAE-ENSMA, Poitiers, France

Abstract. Python is the dominating data science language used in practice, leaving behind other languages like C++, Java and R. Python libraries wrap highly tuned, efficient, accurate C++ and C code for linear algebra, numerical methods and data manipulation. Moreover, the Python runtime works flawlessly across diverse operating systems (Linux, Windows) and CPU architectures, including x86 and ARM. From an accelerator perspective, Python code is processed on multi-core CPUs and GPUs, whose power is not fully exploited. In this paper, we study how to improve Python I/O bottlenecks. We focus on data set summarization to compute a model on a large data set, stored on a CSV file (the most common format used in practice). Heeding these challenges, we introduce two simple, but fundamental, I/O optimizations: parallel multi-threaded read and chunk-based scan (similar to reading file blocks in a DBMS). An experimental validation on different cloud servers, provides a realistic scenario. We show our optimized Python code can work faster than existing Python functions, it exhibits almost linear speed up I/O as more threads are used (up to a limit), but it can still leverage parallel processing for the CPU-intensive floating point computations. To round up our study, we justify chunk size is a critical performance parameter that depends on data set size as well as cloud server configuration.

1 Introduction

Our study aims to highlight I/O on text files as a major bottleneck in AI and ML computations processed by Python [6] in a typical server in the cloud. Most analytic projects target is a predictive machine learning (ML) model. Logistic regression stems from linear regression [4] and logistic regression is a foundation model for deep neural networks [1]. Therefore, we study linear regression (LR) as a representative and fundamental AI model representing a demanding computation (I/O + floating point operations) on a large input matrix, perhaps exceeding main memory limits. Our paper advances [3], where summarization is proposed to accelerate ML computations in the R language, with serial I/O. A major step forward is the parallel computation of summarization in Python, with parallel I/O, with chunked files, in a typical multi-core cloud server. This paper borrows query processing ideas to compute summarization in parallel in a DBMS with SQL queries combining joins and aggregations [5].

2 Accelerating and Scaling Data Summarization

2.1 Definitions: Input Matrix and Machine Learning Model

The input data set is defined as X, a $d \times n$ matrix consisting of a set of n column vectors each with d dimensions. We refer to the machine learning model as Θ, in this case vector $\hat{\beta}$.

Our data summarization works well for multiple ML models Θ such as PCA, NB (Naive Bayes), K-Means clustering, and LR (Linear Regression). Given the mathematical importance of LR, and being used as a theoretical foundation for Neural Networks we focus on its solution.

2.2 Data Set Summarization

In order to compute data summarization matrix Γ for LR, first we augment X with a row of ones to produce a $(d+1) \times n$ matrix $\mathbf{X}$. $\mathbf{Y}$ is an n-row vector corresponding to the output for each n observation. Given a $(d+1) \times n$ input matrix $\mathbf{X}$, and a $(d+1) \times 1$ column vector $\hat{\beta}$ of coefficients, the predicted outputs $\hat{\mathbf{Y}}$, an n-row vector corresponding to the predicted outputs of each n observation produced by the LR model, are computed as $\hat{\mathbf{Y}} = \hat{\beta}^T \mathbf{X} + \epsilon$, where ϵ represents error. For computing Γ, augment $\mathbf{X}$ with $\mathbf{Y}$. In general this $(d+2) \times n$ matrix is defined as $\mathbf{Z}$, but to optimize I/O we create sub-matrix of size $(d+2) \times c$ where c is the chunk size (a block of vectors). Therefore, c can be considered a hyper-parameter and it must be tuned to achieve optimal performance. The serial computation of a model Θ becomes a two-phase algorithm as follows:

- Phase 1: Summarize X: compute $\Gamma = \sum_{i=1}^{n} z_i \otimes z_i^T$;
- Phase 2: Compute model Θ: Solve $\hat{\beta}$ exploiting Γ.

Phase 1: Begin by reading a chunk of the input data set of size $d \times c$. Augment this chunk in main memory with a row vector of c ones and $\mathbf{Y}$ (dependent variable) also a c row vector. This will produce $\mathbf{Z}$, a $(d+2) \times c$, a dense matrix whose size is $\Theta(d^2)$. Compute $\mathbf{Z}\mathbf{Z}^T$ to produce the partial gamma Γ_c.

The sufficient statistics (SS) L, n and Q are defined as follows: $n = |X|$, $L = \sum_{i=1}^{n} x_i$, and $Q = XX^T = \sum_{i=1}^{n} x_i \cdot x_i^T$, n is total number of points in the data set, L is the linear sum of x_i and Q is the sum of vector outer products of x_i. It should be noted that Phase 1 takes most computation time. Phase 2: Exploit the sufficient statistics integrated into the single matrix Γ_c to further compute an ML model. In our target case, the goal is to compute the regression coefficients $\hat{\beta}$ whose solution by the least square method is $\hat{\mathbf{Y}} = \hat{\beta}^T \mathbf{X} + \epsilon$. The "quadratic" sufficient statistic matrix Q and the matrix-vector product XY^T are exploited by substituting them into $\hat{\beta} = (\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{X}\mathbf{Y}^T = Q^{-1}(\mathbf{X}\mathbf{Y}^T)$, where the second expression is much faster to compute because it is based only on Γ.

The serial algorithm has time complexity $\Theta(d^2 n)$ for Phase 1, but $\Theta(d^3)$ for Phase 2 (much lower as $n \to \infty$). That is, Phase 1 dominates time growth.

Data summarization is quadratic in data set dimensionality (i.e. demanding) and therefore, it has a significantly higher time complexity than stochastic gradient descent, the workhorse of AI models.

2.3 Hardware and Software Evolution

Modern computing has evolved to feature multi-core CPUs, with high-end models like the Xeon Granite Rapids offering up to 128 cores and support for vectorized instructions, significantly enhancing computational efficiency. Server memory configurations range from 16 to 32 GB, scaling up to terabytes to meet the demands of Big Data and Deep Learning. Advances in storage, with SSDs providing 3X, NVMs providing 10X, faster read speed than HDDs, have significantly narrowed the performance gap between RAM and secondary storage, though I/O from secondary storage remains a bottleneck, albeit less so with modern PCI-connected NVMs.

Processing of Python code is done by multi-core CPUs supporting vectorized instructions, which accelerate matrix computations, but only in memory. In systems without distributed memory (i.e. a cluster of computers), multi-threaded processing is the primary parallelization strategy, involving "coarse" threads (e.g., Python) mapped to hardware CPU threads (embedded in the chip). This setup emphasizes the importance of multi-threading processing for parallel execution in shared RAM scenarios and the alignment of software with hardware threads to optimize efficiency. However, practical limitations such as Python's Global Interpreter Lock (GIL) create a need for alternative parallelism, highlighting the complexity of optimizing parallel processing, especially for I/O operations.

2.4 Parallel I/O Efficient Algorithm

Let p be the number of processors (cores, machines), under a partitioned memory architecture, where each processor has its independent main memory and persistent storage. We assume $d \ll n$ and $p \ll n$, but d is independent from p. Our parallel algorithm follows:

– Data set X is uniformly partitioned among the p processors with $\approx n/p$ points x_i per partition. Initialize: $\Gamma = [0]$
– Phase 1 in parallel with p processors: Compute Γ in parallel, dynamically building z_i in main memory, updating Γ_j in main memory reading X_j from persistent storage in blocks. Send the partial matrices to the master processor and aggregate the p partial summary matrices into the global matrix Γ.
– Phase 2 at master processor: Compute Θ using Γ in intermediate computations. Iterate method until convergence exploiting Γ in intermediate matrix computations.

Notice we opt for a partitioned memory model with parallel I/O [2] instead of PRAM, which would require locking mechanisms in Phase 1, introducing significant overhead. Parallelization is accomplished by computing separate summaries on each partition. That is, threads do not share variables in main memory. The main reason this parallelization is feasible is because matrix multiplication is distributive and additive, meaning we can compute separate matrix multiplications and add them at the end (with negligible overhead to lock the global summarization). Phase 2 is very fast since it does not depend on n.

3 Experimental Evaluation

As explained in Section 2 Phase 1 takes around 99% of time and Phase 2 takes only 1%-2%. Therefore, we focus on studying Phase 1, which is I/O bound.

3.1 Evaluation Setup

Cloud Server Configuration We ran experiments on three cloud servers, provided by Amazon AWS. Our results should be similar on other cloud providers, which also use VMware for virtualization. Our server specifications were as follows: Server 1 (2 vCPUs) is a Xeon-based HVM domU instance with an Intel(R) Xeon(R) E5-2686 v4 CPU, 2 cores per socket - 1 socket - 1 thread/core, 8GB RAM. Server 2 (4vCPUs), an Amazon EC2 g4dn.xlarge instance, with an Intel Xeon CPU with 24 cores and 48 physical threads, 16 GB RAM. 16 GB RAM, 2 cores per socket - 1 socket - 2 threads/core. Server 3 (8 vCPUs) is an Amazon EC2 g4dn.xlarge instance, with an Intel Xeon CPU with 18 cores and 36 physical threads, 32 GB RAM. All servers had detached SSD storage, connected via a fast network, with limited space around 200 GBs and they were running Linux Ubuntu under VMware.

Python Data Science Libraries Our prototype setup leverages Python libraries: `Pandas` for I/O operations, `NumPy` for numerical computations, and `threading` for multi-threaded parallelism to efficiently read large data sets, eliminating RAM limitations. Data sets were chunked (divided into blocks similar to an SQL table) and partitioned across p files for scalable processing: parallel multi-threaded I/O, lock-free, without RAM limitations. Each thread worked in parallel computing Γ_c matrices. As explained above, to mitigate the I/O bottleneck, we utilize Python's built-in `threading` library to spawn p threads equal to the number of data set partitions, stored on p files. Notice we also exploit CPU parallel processing for floating point operations.

3.2 Profiling Computation Steps to Identify Bottlenecks

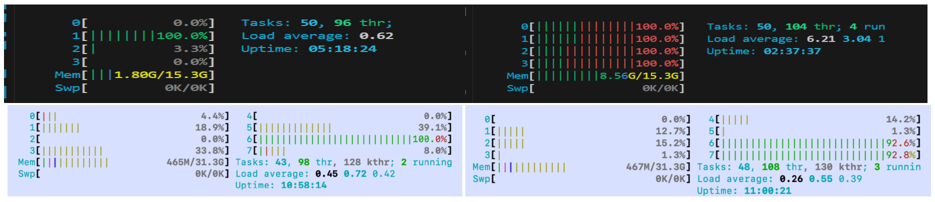
To verify I/O is the main bottleneck in the computation, we profiled each computation step on three widely different cloud servers, as shown in Table 1. We analyzed how the number of threads affects speed. Table 1 shows that 90% of the whole computation time is due to I/O and this fraction is bigger for large n, when the data set size exceeds RAM size. Increasing the number of threads accelerates the summarization computation, but to a limit: for the smaller cloud servers (2 vCPUs, 4 vCPUs) 8 threads fail, whereas for the larger server (8 vCPUs) the speedup stops at 2 threads. These time results indicate there is a complex interaction among number of vCPUs, number of threads, data set size and chunk size. Unfortunately, we could not determine the specific storage device information (model, block size, PCI vs SATA connection).

Table 1: Profiling each step, highlighting significant time is spent on I/O.

	#threads	size	chunk size	**I/O**	summarization	total	total
vCPUs	p	n	c	time	time	time	speedup
2	1	16M	100k	491	13	504	1.0X
2	2	16M	100k	242	10	252	2.0X
2	4	16M	100k	173	6	179	2.8X
2	8	16M	100k	173	fail	fail	-
4	1	16M	100k	421	10	431	1.0X
4	2	16M	100k	219	7	227	1.9X
4	4	16M	100k	165	9	174	2.5X
4	8	16M	100k	165	fail	fail	-
8	1	1M	1k	30	8	38	1.0X
8	2	1M	1k	22	4	26	1.5X
8	4	1M	1k	275	3	278	0.1X
8	8	1M	1k	885	2	887	0.1X

3.3 Accelerating I/O Speed with Parallelization and Chunks

Our next experiments aim to find out how many threads "saturate" the CPU cores. Figure 1 compares CPU core saturation with 1 thread versus the optimal number of threads, obtained from Table 1. That is, we want to use cores as close as possible to 100% to avoid cores, and vCPUs in consequence, being idle. As can be seen, 4 threads are optimal for the 4 vCPU server, but only 2 threads are optimal for the 8 vCPU server. Notice the virtualization software and the operating system consume CPU cycles anyway on some CPU cores.



```
    0[                 0.0%]   Tasks: 50, 96 thr;
    1[||||||||||100.0%]   Load average: 0.62
    2[|                3.3%]   Uptime: 05:18:24
    3[                 0.0%]
  Mem[|||1.80G/15.3G]
  Swp[                0K/0K]
```

```
    0[||||||||||||||100.0%]   Tasks: 50, 104 thr; 4 run
    1[||||||||||||||100.0%]   Load average: 6.21 3.04 1
    2[||||||||||||||100.0%]   Uptime: 02:37:37
    3[||||||||||||||100.0%]
  Mem[|||||||||8.56G/15.3G]
  Swp[                 0K/0K]
```

```
    0[|||                          4.4%]   4[                               0.0%]
    1[|||||||                     18.9%]   5[|||||||||||||                 39.1%]
    2[                             0.0%]   6[||||||||||||||||||||||||||||100.0%]
    3[|||||||||||                 33.8%]   7[|||||                          8.0%]
  Mem[||||||||||||           465M/31.3G]   Tasks: 43, 98 thr, 128 kthr; 2 running
  Swp[                            0K/0K]   Load average: 0.45 0.72 0.42
                                           Uptime: 10:58:14
```

```
    0[                             0.0%]   4[|||||                          14.2%]
    1[|||||                       12.7%]   5[|                               1.3%]
    2[|||||                       15.2%]   6[|||||||||||||||||||||||||||||92.6%]
    3[|                            1.3%]   7[|||||||||||||||||||||||||||||92.8%]
  Mem[||||||||||||           467M/31.3G]   Tasks: 48, 108 thr, 130 kthr; 3 runnin
  Swp[                            0K/0K]   Load average: 0.26 0.55 0.39
                                           Uptime: 11:00:21
```

Fig. 1: Achieving full vCPU utilization with multi-threaded I/O (black=4 vCPUs, gray=8 vCPUs): left/up=1 thread, 4 vCPUs; right/up=4 threads, 4 vCPUs; left/down=1 thread, 8 vCPUs; right/down=2 threads, 8 vCPUs.

Our last experiments explore chunk size on two cloud servers, shown in Figure 2. For the 4 vCPUs server increasing c decreases time, but the impact is not look significant. A large chunk size c =100k gives optimal time, but it does not improve when chunk size c approaches $n = 1M$. On the other hand, a smaller chunk size around c =100 is best for the 8 vCPUs server and then performance decreases as n grows. These results indicate there exists a chunk size that minimizes I/O time and they highlight that d, n alone are not sufficient to determine chunk size. Why? because the number of vCPUs and the storage device also have an impact.

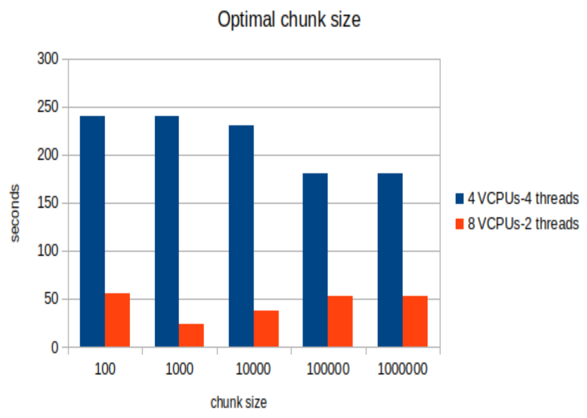


Fig. 2: Finding optimal chunk size on 2 cloud servers with their best number of threads: 4 vCPUs and 4 threads; 8 vCPUs and 2 threads.

References

1. Charu C. Aggarwal. *Neural Networks and Deep Learning - A Textbook*. Springer, 2023.
2. Michael A. Bender, Gerth Stølting Brodal, Rolf Fagerberg, Riko Jacob, and Elias Vicari. Optimal sparse matrix dense vector multiplication in the i/o-model. In Phillip B. Gibbons and Christian Scheideler, editors, *SPAA 2007: Proceedings of the 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Diego, California, USA, June 9-11, 2007*, pages 61–70. ACM, 2007.
3. Siva Uday Sampreeth Chebolu, Carlos Ordonez, and Sikder Tahsin Al-Amin. Scalable machine learning in the R language using a summarization matrix. In *Database and Expert Systems Applications DEXA*, pages 247–262, 2019.
4. T. Hastie, R. Tibshirani, and J.H. Friedman. *The Elements of Statistical Learning*. Springer, New York, 1st edition, 2001.
5. Carlos Ordonez. Scalable parallel machine learning computing a summarization matrix with SQL queries. In *IEEE Big Data*, pages 151–160, 2017.
6. Dipanjan Sarkar, Raghav Bali, and Tushar Sharma. Practical machine learning with Python. *A Problem-Solvers Guide To Building Real-World Intelligent Systems. Berkely: Apress*, 2018.