QuantCloud: A Software with Automated Parallel Python for Quantitative Finance Applications

Peng Zhang
Applied Mathematics Department
Stony Brook University
NY 11794, United States
Peng.Zhang@Stonybrook.edu

Yuxiang Gao
Midea Emerging Technology Center
CA 95134, United States
Yuxiang1.Gao@Midea.com

Xiang Shi
Advanced Risk & Portfolio Management (ARPM),
NY 10023 United States
Xiang.Shi@arpm.co

***Abstract*—Quantitative Finance is a field that replies on data analysis and big data enabling software to discover market signals. In this, a decisive factor is the speed that concerns execution speed and software development speed. So, an efficient software plays a key role in helping trading firms. Inspired by this, we present a novel software: QuantCloud to integrate a parallel Python system with a C++-coded Big Data system. C++ is used to implement this big data system and Python is used to code the user methods. The automated parallel execution of Python codes is built upon a coprocess-based parallel strategy. We test our software using two popular algorithms: moving-window and autoregressive moving-average (ARMA). We conduct an extensive comparative study between Intel Xeon E5 and Xeon Phi processors. The results show that our method achieved a nearly linear speedup for executing Python codes in parallel, prefect for today's multicore processors.**

***Keywords*—Quantitative Finance Software, Parallel Python, Big Data, Cloud computing.**

I. Introduction

Quantitative Finance is a field that extends mathematical models to the finance problem thus it is also known as computational finance. In this field, the revolution of computational technologies has been shaping the best practice and future of quantitative finance. In the high-frequency trading age, a program trading system is developed to use powerful computers to transact a large number of orders as quickly as possible. The whole order and withdraw process may happen in a microsecond level or even less [1]. However, as the age of big data arrives, the science, social and economic including quantitative finance have been undergoing a fierce yet great revolution. In the past, the high-frequency traders had been pursuing a high speed between exchanges for facilitating the buying and selling of shares, currencies and other assets [2, 3]. At present, the finance firms want to compete on strategies as the race for transacting speed among high-frequency traders hit peak [4, 5]. Traders are building more complicated data analysis models to derive deeper profitable signals out of the big finance datasets [6]. Thus, there is a need to construct a novel software that allows fast-developing and fast-testing strategies. This need inspires this work.

Speed is always a decisive factor in maintaining a finance firm's competitive advantage but its meaning is extending. To have a transaction speed without prediction is of no practical value. Currently, the speed concerns with not only the execution speed of a big data analysis model but also the software development speed of a complicated mathematical model. In the practice of this field, Python is the most preferred high-level programming language as it requires fewer lines of code and also has wide availability of statistics libraries in timeseries analysis. On the other hand, C++ is the most ideal language to implement the big data infrastructure system that is able to handle massive amounts of market data as it provides high speed of execution. Considering these facts, we develop an integration system that combines a C++-based big data infrastructure and an automated parallel Python system. As being applied to quantitative finance, this system handles the timeseries market information by this big data infrastructure and meanwhile performs timeseries analysis models that are coded in Python.

Revolutionary technological advances have been stimulating evolutionary industrial adaptation. In this trend, the quantitative finance is a grand pioneer for adapting advanced technologies such as novel multicore processor architectures. Of these, Intel Xeon Phi processor, codenamed as Knights Landing (KNL), is a representative of modern multi-core processors. Different from its former processor families, Intel's KNL has a higher density of processor cores and thus it is optimized more for highly parallel workloads. However, fully exploiting the power of such kind of high-density multi-core processor is by no means a trivial challenge. This needs an effective yet agile way to bring a degree of parallelism to the execution of programs. To this end, we develop a coprocess-based parallel execution for Python.

Contribution synopsis: the main contribution of this work is the design of a software suite, QuantCloud that combines a big data infrastructure with an automated parallel Python system. We also conducted an extensive application-level comparative study using commodity hardware. The results show the efficacy of this software and characterize all essential aspects of performance such as the wallclock time, the speedup and parallel efficiency for the codes in Python, the tick-level latency for commonly-used QF applications on real-world market data.

II. Background and Motivation

A. Big Data in Quantittative Finance

Big data is becoming a critical issue in finance, particularly the quantitative finance with multiple applications, wider usage, given advances in enabling technologies [7]. Big data in finance has covered all principle interests of Big Data such as the data volume, velocity and variety [8-10]. Data volume of market information has been ever increasing at a tremendous rate. For instance, the total shares changed hand is tenfold of 20 years ago and the total number of transactions is increased by 50 times, with this number being more than 120 times during the financial crisis [8]. The prevalence of high-frequency trades (HFTs) has spurred up growth of high-speed data in trading activities. For example, about 70% of the U.S. equity trades are computer driven [10].

B. Python for Quantitative Finance

In daily practice of most trading systems, Python is the most preferred language. For example, Quartz is Bank of America Merrill Lynch's integrated trading, position management, pricing and risk management platform and its entire tech stack uses Python. Athena is J.P. Morgan's next-generation pricing, risk management, analysis and trade management platform, and is a Python-based rapid development environment. Meanwhile, a compiled language C++ is used for the high-performance core of this system, while Python is used for building logic and apps. Python is becoming more and more popular for being easier to use and faster to program than traditional languages including the C++ programming language. So far, there's been a huge spike in demand for Python in the investment banks including Bank of America and J.P. Morgan that are using Python to replace historic legacy systems built in Java/C++. From a practical perspective in quantitative finance, we choose Python as a language to program timeseries analysis algorithms and models on the finance big data.

C. Python Limitation

CPython is the default and most widely-used interpreter for the Python programming language. It is written in C and offers rich extensions with several languages including C. In CPython, global interpreter lock, or GIL, is a mutex lock that prevents concurrent executions of multiple native threads within one process [11]. In other words, Python is implemented in such a way that only one thread can be accessing the interpreter at a time. The exceptions are few: for example, while a thread is waiting for I/O, the interpreter is released so other threads can run [12]. In this literature, the GIL becomes a key limitation in multithreading with Python. As usual, multithreading actually performs worse than serial code [13, 14]. However, the GIL is necessary because CPython's memory management is not thread-safe. A solution to this issue is to use multiple full processes instead of threads [15], where each process uses its own GIL. To overcome this limitation, we present a coprocess-based approach and bring parallel performance to Python code. Sophisticated parallelism and workflow management is hidden in QuantCloud system.

D. High-Density Multicore Processors

Technology has been shaping financial markets so much, that the traders are competing for the fastest equipment rather than the transaction itself. The heart of modern quantitative finance is to reduce the execution time of more complicated models by using more advanced machines. In this battlefield for speed, the processor plays a key role. Simply, a more powerful processor makes the analytics algorithm execute quicker so there is more room to crunch more data and harness more complex models while without sacrificing time. Faster computing means more doing. This is of practical interest to time-critical applications in the field. In today's processor markets, Intel's 2nd-generation Xeon Phi processor, codename as Knights Landing (KNL) [16, 17], is a novel high-density multicore processor and it has 64~72 cores per processor, optimized for a highly-parallel application.

III. System Design

Our system incorporates two parts: a big data infrastructure system and its integration interface with Python. The overview of this integrated system is shown in Figure 1. The design of this big data infrastructure system is extension of our previous work [18]. In this system, an appropriate embedded Python interface is built for effortless integration with this big data infrastructure. Data communication between the main C++ program and embedded Python scripts is through a shared memory system. The Python script is used as a high-level language to program the sequential execution of an algorithm. The coprocesses that execute a code in Python seem to run sequentially instead of parallel and they are transparent at the user-application level. There have been no code changes in the Python scripts so this "as-is" embedding approach supplies a simple yet efficient method to use Python embedded in a C++ program for a complex big data application.

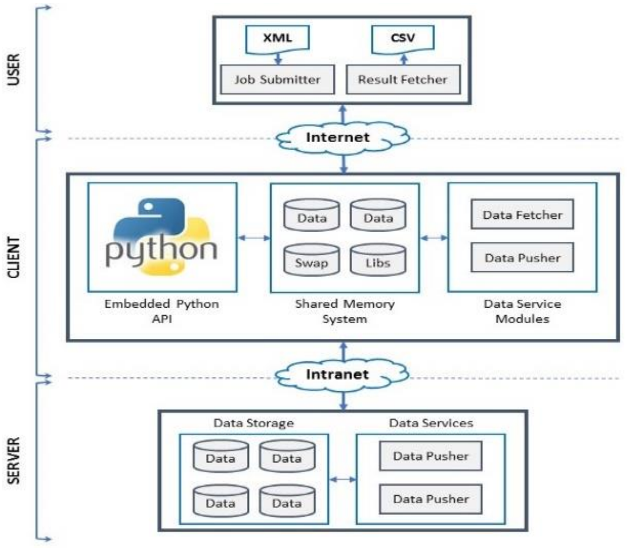

Figure 1. Overview of the integrated Big Data and Parallel Python architecture in the QuantCloud suite

A. Big Data Software Infrastructure

This Big Data infrastructure includes three components: User, Client and Server, in Figure 1. User part is an XML-script portal that is able to receive an application-user job and return CSV-format results to end users. Client part is a platform that executes the computing jobs. It parses a user job, queries the required data from the Server and conducts the job. The Server part is a platform that provides data-centric services. This is a distributed application architecture and adopts a sever-client model that partitions jobs and data between the Client and the

Server. In this system, Client and Server are the provider of computing and data services and User is a service requester. To communicate, Internet communication is between user and client, and Intranet is between server and its clients.

This system enables Cloud platforms as providers for finance big data analytics. Essence of cloud computing is Everything-as-a-Service. In this field, Server may reside on a Storage-as-a-Service provider operating on a cost-per-byte-stored and cost-per-byte-transferred basis. Client may use an Infrastructure-as-a-Service provider provisioning scalable computing resources and operating on a pay-per-use basis. Meanwhile, the finance big data analytics algorithms and models could be supplied as Software-as-a-Service in the Client and defined as pay-per-use software. Simply, User just runs a light-weight kernel thus it is able to be operated on ultra-portable devices. This design helps the big data analytics research products to quickly enter the market with the advent of cloud computing technologies.

The Server part manages the historical market information such as stock transactions. The market information is organized as multiple timeseries and indexed by its date and stock symbol. Here, data is first compressed then hashed before stored on the storage. Data compression is for saving space and hashing for security reason. In addition to data storage, Server responds to timeseries queries [18]. Before querying data, a Client needs to register a Server and establishes a link between data provider (Server) and data requester (Client).

The Client part responds to users' requesters and processes user jobs. Specifically, a user job describes: (1) the requested data information, such as data duration, message type and stock symbols; (2) big data analytics models, such as moving-window timeseries analysis and autoregressive moving-average models; and (3) user-specific analytics codes in Python. Upon arrival of a user job, the Client parses the requested data information then queries timeseries data from its Server. Data analytics starts as soon as the queried data streams flow into this Client node. Only timeseries methods take into account possible internal structure on the market data streams. In Section 5, we would present two popular analytics methods: moving-window analysis method of financial timeseries data and autoregressive moving-average (ARMA) model. The user-specific analytics code is embedded to apply an analytics method on the managed timeseries. Here, our focus is to enable embedded-Python API that allows finance engineers to: (1) easily implement a method in Python and (2) effortless integrate their method with this big data system for ultra-fast low-latency execution. The detail of integration with Python is presented in the following section.

B. *Automated Parallel Python Software System*

Transparency is of essence in the design of embedded parallelized Python APIs in this big data infrastructure system. So, the Python script that is embedded stays unchanged and is integrated "as-is". Figure 2 shows the integration flowchart of the codes in Python in the QuantCloud system. It adopts a coprocess-based mechanism. A parent process stands for a thread of main process and manages the timeseries data streams. At its inception, it spawns a child coprocess that is able to offload its workload. Communication between a parent and its child coprocess is using a parent-child shared memory. A shared memory is attached and its associated parent-child synchronization channel is established at the same time. To execute a code in Python, the child coprocess serializes the timeseries data in the C++ environment, transfers serialized data packets to the Python environment where serialized data is deserialized and re-formatted as data structures in Python. This completes data conversion from C++ to Python. The interpreter is called to execute the script as long as data is ready to use. The results from the Python code are serialized then returned to the child coprocess. The child coprocess deserializes data packets then restores structures. The results are finally transmitted back to memory space of the parent process and then an acknowledge signal is sent to this parent process upon completion of this job.

The code in Python is executed in a single-thread model. A child coprocess is operating in a multi-threaded asynchronous model. It has a built-in job queue able to buffer multiple jobs at the same time and executes the buffered jobs as first-in first-out (FIFO). It operates three threads: thread 1 is for messaging with a parent process; thread 2 for executing the code in Python and thread 3 for deserializing results from the Python environment. This asynchronous execution mechanism, though complicates the implementation as requires more thread-safe codes, could effectively overlap the data serialization and the data analytics operations, thus it helps reduce the latency that is caused by the extra serialization operations. Optionally, an additional thread is configured to monitor the health state of the parent process periodically and provide fault tolerance. Particularly, it performs a safe shutdown when a failure is detected. Otherwise, an orphan process appears at occurrence of program faulty and error.

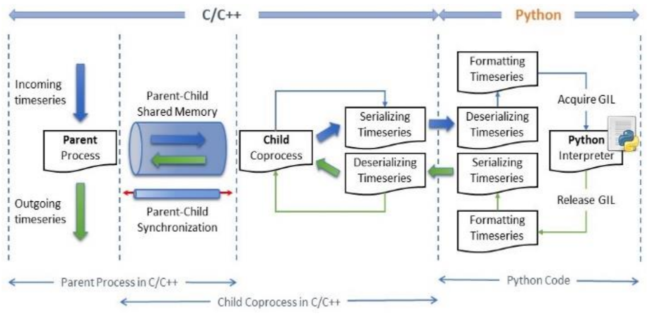

Figure 2. Integration of automated parallel Python system in the big data infrastructure in the QuantCloud software suite

IV. PROTOTYPE IMPLEMENTATIONS

We build a prototype to study the performance characteristics of this proposed system. In this section, we present the software stack about the prototype implementation and the hardware that we use to benchmark this prototype. In next section, we present the finance big data analytics models that use this prototype for test and describe market data and performance measurements.

A. *Software Stack*

The prototype is coded in C++. The input script is in XML format and result is reported as in CSV file. The communication among User, Client and Server uses the TCP/IP protocol. The database on the Server and the query of timeseries data on the

Client follow our previous work [18]. The automated parallel Python API is provided on the Client.

Within one Client instance, the multithreaded programming is used for intra-node parallelism on shared memory. In this, the thread pool is used to manage the threads. The Python code is run in a coprocess that is referred to as child in Fig. 2. To interact with the main process, a memory segment is shared among the parent (the main program) and its child (the coprocess). This addresses the data transfer between the parent and its child. This child is responsible to interact with Python. The workflow of a child is as follows: the incoming timeseries is serialized and transferred to Python; in Python, serialized timeseries is restored and reformatted as Python data types; then GIL is acquired to conduct the code in Python; last the produced result is reformatted, serialized and transmitted back to the child where the result is returned to its parent's memory. Upon task accomplishment, the child sends an acknowledgement signal to its parent. This completes the work cycle of a child coprocess.

B. Hardware Platform

System 1: Dell PowerEdge R720, installed with two Intel Xeon E5-2603 processors at 1.8 GHz; a total of eight cores per server; 32 GB DDR3 RAM and 500 GB SATA hard drive. Max memory bandwidth is 34.1 GB/s. In this system, operating system is CentOS Linux 7.3 and compiler is GCC 4.8.5. This processor launched on Q1'12 and already discontinued at Q2'15 so it represents a legacy processor.

System 2: QCT QuantaPlex S41T-2U4N system. Each node has one Intel Xeon Phi 7230F processor, codenamed Knights Landing (KNL). Each KNL has a total of 64 cores (1.3 GHz); 128 GB DDR4 RAM and 1 TB SATA SSD. Max memory bandwidth is 115.2 GB/s. In this system, operating system is Red Hat Enterprise Linux 7.2 and compiler is ICC 17.0.1. This processor launched on Q4'16 and it features a high-density compute optimized solution.

V. EXPERMENTS

We experiment the real-world financial analysis models and market tick datasets using this prototype. In this section, we first introduce the moving-window analysis and the autoregressive moving-average (ARMA) analysis of financial timeseries data. Then we describe the financial market tick datasets that we use to conduct the tests. Last, we describe the measurement in tests.

A. Financial Analysis Models

1) Moving-Window Analysis

A moving window method in financial timeseries is one of the most common approaches in many models [19]. We hereby simulate this approach by coding the user-specific data-process functions in Python. The input timeseries stream of tick data is managed by the big data infrastructure, including querying the timeseries, computing the logarithmic returns and formatting a fixed-length window. After these steps, the preprocessed data streams flow into downstream embedded-Python nodes where the Python-coded functions are applied to the data. Final result is exported in CSV files. In this test, two data process functions: 'abs' and 'mean' are programmed in Python.

The flowchart for these tests is shown in Figure 3. Among these, we test three scenarios: (a) "1-node abs": a single embedded-Python node is added to find absolute values of logarithmic returns; (b) "1-node abs + mean": a single embedded-Python node is added to find the average of absolute logarithmic returns; (c) "2-node abs + mean": two embedded-Python nodes are added, where first node is added for finding absolute values of logarithmic returns and second node for finding the average of absolute values from first node. Actually, the result of case (c) is the same as that of case (b) and the difference is number of embedded-Python nodes.

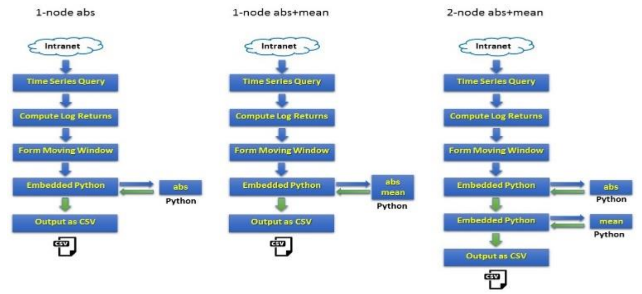

Figure 3. Flowchart for moving-window analysis: 1-node abs (left), 1-node abs+mean (middle) and 2-node abs+mean (right).

2) Autoregressive Moving Average (ARMA)

Outlier detection and data cleaning is the first must-have step of most financial modeling pipelines [7, 20]. We hereby test the autoregressive moving average (ARMA) model on the financial timeseries data. The ARMA model is a tool to understand the values of timeseries. Same to previous study, the input dataset is the tick data for S&P500 trade. The output is the result of the ARMA model within certain fixed-length window of financial timeseries. In this test, the ARMA model is coded in Python. In addition, the Hampel method [21, 22] is coded in Python and is used for truncating the outliers. Similarly, data flow is managed by this big data infrastructure. That is, the raw market tick data is preprocessed before entering the Hampel and AMRA nodes. The flowchart is in Figure 4. In this test, two Python nodes are inserted in this process pipeline.

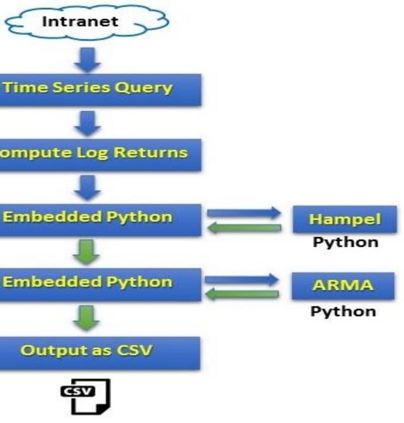

Figure 4: Flowchart for ARMA with Hampel method

B. Performance Measurements

Wallclock time is the amount of elapsed time from the start to the completion of a process pipeline, including the time that queries data from database, prepares timeseries, computes logarithmic returns, executes Python scripts and writes results to disk. This timing result is called as overall wallclock time. Meanwhile, we measure the cumulative time that is spent in executing the codes in Python. In the practice, the time spent in executing a code in Python is the elapsed time from entering to leaving the Python code. This measurement is done at the child coprocess side and includes the delay of C-Python API. This timing result is called as embedded Python wallclock time. The time measure is with a resolution of microseconds.

Latency is reported in microseconds per tick and represents the amount of time elapsed for processing a single tick message on average. It is computed as the overall wallclock time divided by the total number of tick messages. *Speedup* for the codes in Python is defined as a ratio of $T(1)$ over $T(n)$, where $T(1)$ and $T(n)$ are the elapsed times of 1 and n child coprocesses for the codes in Python: $S(n) = T(1)/T(n)$. *Parallel efficiency* is $E(n) = S(n)/n$. This speedup ratio $S(n)$ and parallel efficiency $E(n)$ is used to provide an estimate for how well this embedded-Python system speeds up. It is used to generate a plot of the elapsed time vs. the number of coprocesses and to understand the behavior of the parallelized Python scripts on multicore processor architectures.

VI. Results and Discussion

A. *Performance on Intel Xeon E5-2603 CPU*

1) Moving-Window Analysis

Figs. 5 and 6 present the overall wallclock time (in second) and the latency (in µs/tick), respectively, for processing S&P500 trade tick data for 7 days. These results show that: Our approach brings great parallel performance to the codes in Python. In "1-node: abs" case, wallclock time is reduced from 177 to 56 sec and performance is improved by 68% (Fig. 5) and the latency is reduced to 0.82 µs/tick (Fig. 6). Calling more functions is more expensive. For example, the performance of the "1-node: abs+mean" case is ever worse than that of the "1-node: abs" case (Fig. 5). The former calls two Python functions and the latter calls only one once.

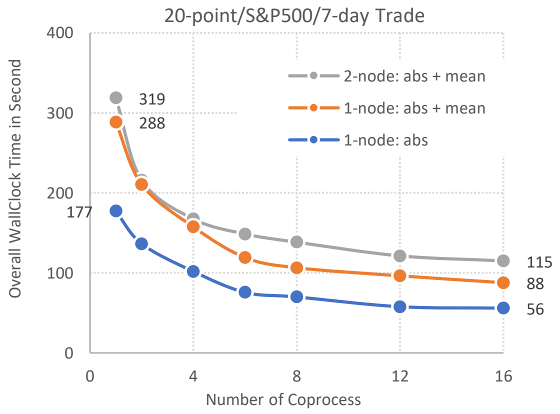

Figure 5. Wallclock time in seconds for processing trade tick data of S&P500 stocks in 7 days (window period: 20)

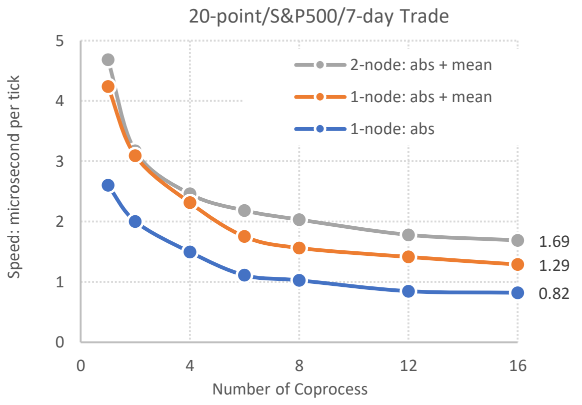

Figure 6. The latency in µs per tick for processing trade tick data of S&P500 stocks in 7 days (window period: 20)

2) ARMA model

Figs. 7 and 8 present the overall and embedded-Python wallclock times (in seconds), respectively. In this test, we tested two cases: 16-stock and 8-stock. From results, this test affirmed the parallel performance our approach brings to the code in Python. As the number of coprocesses increases, the overall and the embedded-Python wallclock times are reduced in Figs. 7 and 8. The overall performance for 16-stock and 8-stock are improved by 51% and 43%, respectively (Fig. 7). The optimal number of coprocesses for the 16-stock case is 8 (Fig. 7).

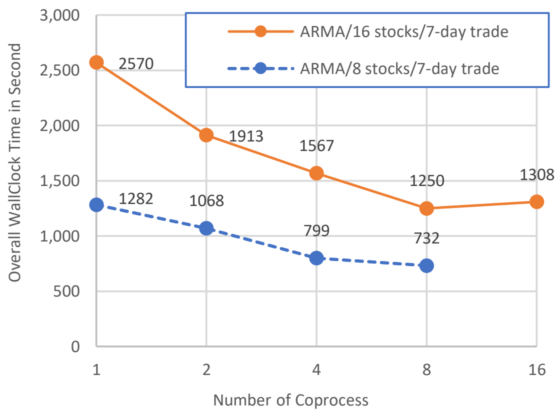

Figure 7. Overall wallclock time for the ARMA model.

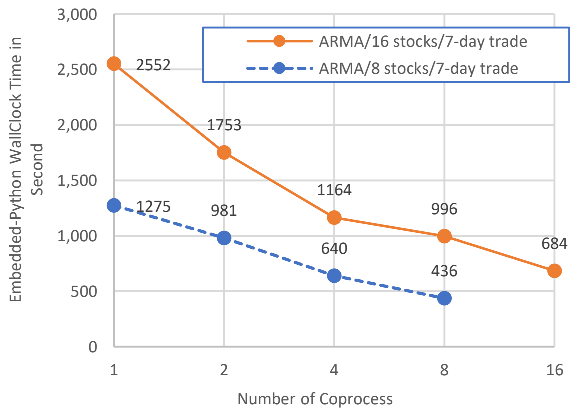

Figure 8. Embedded-Python wallclock time for ARMA.

B. Performance on Intel Xeon Phi 7230F

1) Moving-Window Analysis

Fig. 9 presents the overall and embedded-Python wallclock times. This test reaffirmed the scalability of our approach. With the increase of coprocesses, the performance is consistently improved and the speedup for Python codes is almost linear.

2) ARMA model

We use 64 stocks and 7-day trade in this test. Fig. 11 presents the overall wallclock time vs. the number of coprocesses. In this figure, both axes are in logarithmic scale to clarify the scalability. This test restated that our platform could bring significant performance improvement on this time-consuming ARMA model: the wallclock time is reduced from 6 days (21,008 seconds) to 20 minutes (1,183 seconds) in Fig. 11. The Python part is speeded up by a factor of 28.4. In most configurations, the parallel efficiency in the Python part is no less than 90%. Optimal number of coprocesses is 32 for this test.

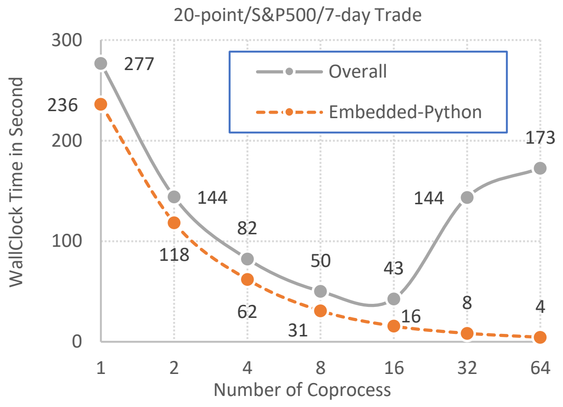

Figure 9. Overall and embedded-Python wallclock time for processing trade tick data of S&P500 stocks for 7 days.

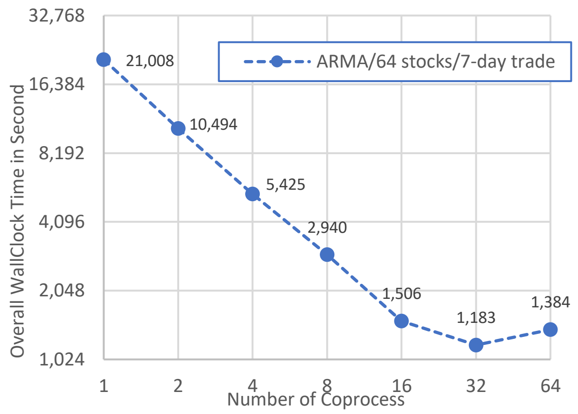

Figure 10. Overall wallclock time in second for the ARMA model on Intel Xeon Phi processor.

C. Intel Xeon E5 vs. Intel Xeon Phi 2

We compare two Intel Xeon processor architectures at the finance big data analytics level in this subsection.

1) Moving-Window Analysis

In this test, we choose the "2-node: abs+mean" case as it is the most time-consuming case among three cases. Wallclock time is shown in Fig. 12. This comparison shows that Intel Xeon Phi processor consistently outperforms Intel Xeon E5 processor. The former is 2.67 times faster than the latter in this test. Our method improved the overall performance by 64% on Xeon E5 CPU and 84% on Xeon Phi CPU. The latency is reduced to 1.69 µs/tick on Xeon E5 and 0.63 µs/tick on Xeon Phi. The latter's latency is only 1/3 of the former processor model.

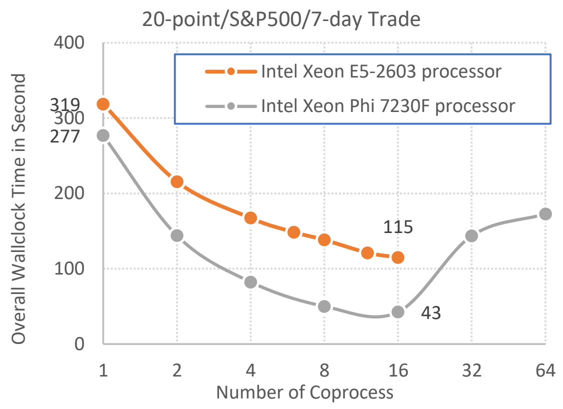

Figure 11. Overall wallclock time for processing S&P500 trade data for 7 days, using 2-node: abs and mean.

2) ARMA model

We compare the performance of ARMA model on Xeon E5 and Xeon Phi processors in two test sets as follows.

Test 1: we test the same problem size: a total of 16 stocks and 7-day trade tick market data. The overall clock time vs. the number of coprocess is shown and compared in Fig. 12.

This comparison shows when a small amount of coprocesses are used, Intel Xeon Phi processor actually performs worse than Intel Xeon E5. For example, when the Python code runs in a single process mode (i.e., number of coprocess is one), it takes Xeon Phi 5262 seconds, while Xeon E5 needs only 2570 seconds that is ~2x faster than the Xeon Phi processor. However, as the number of coprocesses increases, Xeon Phi quickly outperforms Xeon E5 and shows its superiority with aid of massive parallel execution of Python codes. In this test, the Xeon Phi reduces the overall wallclock time to 418 seconds and is ~3x faster than the Xeon E5.

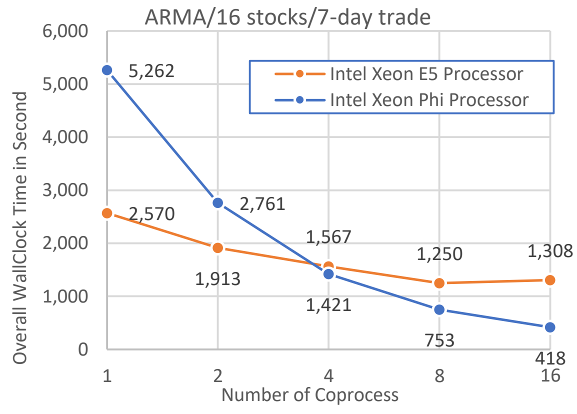

Figure 12. Overall wallclock time of ARMA for 16 stocks and 7-day trade on Intel Xeon E5 and Xeon Phi CPUs.

Thus, it is no coincidence that effective parallel execution of the Python code is essential at exploring the performance of the upcoming multicore platforms such as the Knights Landing and its processor family. In this work, we have already showed an effective coprocess-based parallelism for the codes in Python and their seamless integration with a big data infrastructure.

Test 2: we conduct a scalability test by using two problem sizes: 16 stocks for Xeon E5 and 64 stocks for Xeon Phi. 7-day trade tick data is used for both. In results, the per-stock latency (i.e., second per stock) is presented in Fig. 13. These results show that: Xeon Phi processor consistently outperforms the Xeon E5, with the increase of coprocesses, in terms of the per-stock latency. The Xeon Phi processor excels at lowering the latency than the Xeon E5. The Xeon Phi is able to improve the latency by 95% but the Xeon E5 only by 52%. The Xeon Phi has a higher workload throughput than the Xeon E5. The Xeon Phi is able to process as many as 195 stocks in an hour while the Xeon E5 could just do a maximum of 46 stocks in an hour. In this measure, the Xeon Phi is 4.2x faster than the Xeon E5.

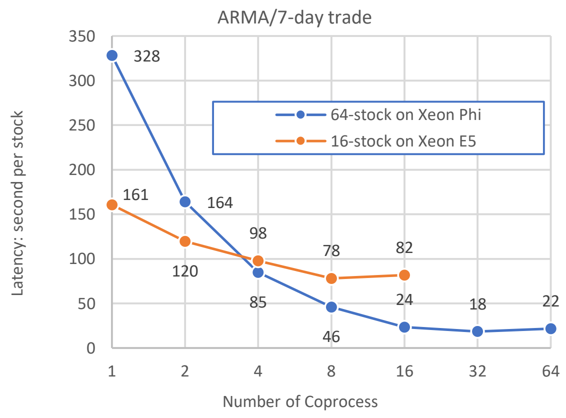

Figure 13. Per-stock latency (second per stock) of ARMA for 16 stocks on Intel Xeon E5 and 64 stocks on Xeon Phi CPUs.

Collectively, by observing these results in Tests 1 and 2, we can see that: (1) Intel Xeon E5 processor is about 2x times faster than Intel Xeon Phi processor when the Python code is simply executed in a single process; (2) the power of a novel Intel Xeon Phi processor cannot be explored until the parallel execution of the Python code is employed. In other words, if the application cannot execute in parallel, it can hardly take advantage of novel modern processors' capabilities and it may end up with an even worse performance. This appears true not only for these finance big data applications in this work but also for a wide range of modern applications [23-25].

D. Summary and Discussion

In this section, we have tested the moving-window analysis and the ARMA model and compared the performance of Intel Xeon E5 and Xeon Phi processors. These results can demonstrate that: our approach, a coprocess-based parallel execution of the code in Python could bring significant parallel performance to all cases. Specifically, the overall wallclock time and the time for the Python code is greatly reduced, as more coprocesses used. The speedup for the Python code is nearly linear and the parallel efficiency is around 90%.

Our approach plays a key role at fully delivering the power of modern high density multicore processors. Clearly, the peak performance of Intel Xeon Phi processor is greatly larger than that of its previous Intel processors. However, fully exploring the power of such modern multicore processors requires highly parallel application programs. In the meanwhile, programming difficulty increases with increased complexity of architectures. With the number of cores in mainstream processors predicted to scale to hundreds today, there is an urgent need to develop a highly-parallel application-specific platform for the real-world application users. To this aim, this work presents an approach that effectively brings coprocess-based parallel execution to the Python while integrates with a finance big data infrastructure. This yields an integrated big data and parallel python platform towards modern quantitative finance applications. This effort is of practical interest in the research and industry.

VII. CONCLUSION

This work presents an integrated big data and parallel Python platform for the quantitative finance applications. This platform incorporates a big data infrastructure system to manage finance timeseries market data and a built-in embedded-Python API to execute the Python code on these managed timeseries streams. The code in Python is able to be executed in highly parallel. We prototype this proposed system and test our prototype with two popular applications and the NYSE tick data. Results show that our system could bring significant parallel performance to all of the cases; the execution time is greatly reduced as the number of processes increases; the speedup for parallel Python is nearly linear and the parallel efficiency for most cases is as high as around 90%. These achievements demonstrated the efficacy of our proposed system model and the efficiency of our software prototype at the real-world applications level.

REFERENCES

[1] J. J. Angel, L. E. Harris, and C. S. Spatt, "Equity trading in the 21st century: An update," The Quarterly Journal of Finance, vol. 5, p. 1550002, 2015.

[2] A. Carrion, "Very fast money: High-frequency trading on the NASDAQ," Journal of Financial Markets, vol. 16, pp. 680-711, 2013.

[3] I. Aldridge, High-frequency trading: a practical guide to algorithmic strategies and trading systems vol. 459: John Wiley and Sons, 2009.

[4] M. A. Goldstein, P. Kumar, and F. C. Graves, "Computerized and High‐Frequency Trading," Financial Review, vol. 49, pp. 177-202, 2014.

[5] J.-P. Serbera and P. Paumard, "The fall of high-frequency trading: A survey of competition and profits," Research in International Business and Finance, vol. 36, pp. 271-287, 2016.

[6] G. Meyer and N. Bullock. (2017, March 30). Race for speed among algo traders hits peak. Available: https://www.ft.com/content/6961129e-14fa-11e7-80f4-13e067d5072c

[7] X. Shi, P. Zhang, and S. U. Khan, "Quantitative Data Analysis in Finance," in Handbook of Big Data Technologies, A. Y. Zomaya and S. Sakr, Eds., ed Cham: Springer International Publishing, 2017, pp. 719-753.

[8] B. Fang and P. Zhang, "Big Data in Finance," in Big Data Concepts, Theories, and Applications, S. Yu and S. Guo, Eds., ed Cham: Springer International Publishing, 2016, pp. 391-412.

[9] M. Peat, "Big data in finance," InFinance: The Magazine for Finsia Members, vol. 127, p. 34, 2013.

[10] T. Seth and V. Chaudhary, "Big Data in Finance," ed, 2015.

[11] D. Beazley, "Understanding the python gil," in PyCON Python Conference. Atlanta, Georgia, 2010.

[12] K. Kinder, "Event-driven programming with Twisted and Python," Linux journal, vol. 2005, p. 6, 2005.

[13] (2015). SciPy Cookbook. url: http://scipy-cookbook.readthedocs.io/

[14] R. Odaira, J. G. Castanos, and H. Tomari, "Eliminating global interpreter locks in ruby through hardware transactional memory," in ACM SIGPLAN Notices, 2014, pp. 131-142.

[15] L. Dalcin, R. Paz, and M. A. Storti, "MPI for Python," J. Parallel Distrib. Comput., vol. 65, pp. 1108-1115, 2005.

[16] Y. Gao and W.-M. Chen, "Family Relationship Inference Using Knights Landing Platform," in Cyber Security and Cloud Computing (CSCloud), 2017 IEEE 4th International Conference on, 2017, pp. 27-30.

[17] C. Zhou, Y. Gao, and W. Howard, "Evaluation of Combining Bootstrap with Multiple Imputation Using R on Knights Landing Platform," in Cyber Security and Cloud Computing (CSCloud), 2017 IEEE 4th International Conference on, 2017, pp. 14-17.

[18] P. Zhang, K. Yu, J. Yu, and S. Khan, "QuantCloud: Big Data Infrastructure for Quantitative Finance on the Cloud," IEEE Transactions on Big Data (in press) doi: 10.1109/TBDATA.2017.2649544, 2017.

[19] N. R. Swanson, "Money and output viewed through a rolling window," Journal of monetary Economics, vol. 41, pp. 455-474, 1998.

[20] S. T. Rachev, S. V. Stoyanov, and F. J. Fabozzi, Risk and Uncertainty vol. 211: John Wiley & Sons, 2011.

[21] R. K. Pearson, "Outliers in process modeling and identification," IEEE Transactions on control systems technology, vol. 10, pp. 55-63, 2002.

[22] H. Liu, S. Shah, and W. Jiang, "On-line outlier detection and data cleaning," Computers & chemical engineering, vol. 28, pp. 1635-1647, 2004.

[23] G. Lawson, M. Sosonkina, T. Ezer, and Y. Shen, "Empirical Mode Decomposition for Modeling of Parallel Applications on Intel Xeon Phi Processors," in Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2017, pp. 1000-1008.

[24] S. J. Pennycook, C. J. Hughes, M. Smelyanskiy, and S. A. Jarvis, "Exploring simd for molecular dynamics, using intel® xeon® processors and intel® xeon phi coprocessors," in Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on, 2013, pp. 1085-1097.

[25] J. Reinders, "An overview of programming for Intel Xeon processors and Intel Xeon Phi coprocessors," Intel Corporation, Santa Clara, 2012.

[1] J. J. Angel, L. E. Harris, and C. S. Spatt, "Equity trading in the 21st century: An update," *The Quarterly Journal of Finance,* vol. 5, p. 1550002, 2015.

[2] A. Carrion, "Very fast money: High-frequency trading on the NASDAQ," *Journal of Financial Markets,* vol. 16, pp. 680-711, 2013.

[3] I. Aldridge, *High-frequency trading: a practical guide to algorithmic strategies and trading systems* vol. 459: John Wiley and Sons, 2009.

[4] M. A. Goldstein, P. Kumar, and F. C. Graves, "Computerized and High-Frequency Trading," *Financial Review,* vol. 49, pp. 177-202, 2014.

[5] J.-P. Serbera and P. Paumard, "The fall of high-frequency trading: A survey of competition and profits," *Research in International Business and Finance,* vol. 36, pp. 271-287, 2016.

[6] G. Meyer and N. Bullock. (2017, March 30). *Race for speed among algo traders hits peak*. Available: https://www.ft.com/content/6961129e-14fa-11e7-80f4-13e067d5072c

[7] X. Shi, P. Zhang, and S. U. Khan, "Quantitative Data Analysis in Finance," in *Handbook of Big Data Technologies*, A. Y. Zomaya and S. Sakr, Eds., ed Cham: Springer International Publishing, 2017, pp. 719-753.

[8] B. Fang and P. Zhang, "Big Data in Finance," in *Big Data Concepts, Theories, and Applications*, S. Yu and S. Guo, Eds., ed Cham: Springer International Publishing, 2016, pp. 391-412.

[9] M. Peat, "Big data in finance," *InFinance: The Magazine for Finsia Members,* vol. 127, p. 34, 2013.

[10] T. Seth and V. Chaudhary, "Big Data in Finance," ed, 2015.

[11] D. Beazley, "Understanding the python gil," in *PyCON Python Conference. Atlanta, Georgia*, 2010.

[12] K. Kinder, "Event-driven programming with Twisted and Python," *Linux journal,* vol. 2005, p. 6, 2005.

[13] (2015). *SciPy Cookbook*. Available: http://scipy-cookbook.readthedocs.io/

[14] R. Odaira, J. G. Castanos, and H. Tomari, "Eliminating global interpreter locks in ruby through hardware transactional memory," in *ACM SIGPLAN Notices*, 2014, pp. 131-142.

[15] L. Dalcin, R. Paz, and M. A. Storti, "MPI for Python," *J. Parallel Distrib. Comput.,* vol. 65, pp. 1108-1115, 2005.

[16] Y. Gao and W.-M. Chen, "Family Relationship Inference Using Knights Landing Platform," in *Cyber Security and Cloud Computing (CSCloud), 2017 IEEE 4th International Conference on*, 2017, pp. 27-30.

[17] C. Zhou, Y. Gao, and W. Howard, "Evaluation of Combining Bootstrap with Multiple Imputation Using R on Knights Landing Platform," in *Cyber Security and Cloud Computing (CSCloud), 2017 IEEE 4th International Conference on*, 2017, pp. 14-17.

[18] P. Zhang, K. Yu, J. Yu, and S. Khan, "QuantCloud: Big Data Infrastructure for Quantitative Finance on the Cloud," *IEEE Transactions on Big Data (in press) doi: 10.1109/TBDATA.2017.2649544,* 2017.

[19] N. R. Swanson, "Money and output viewed through a rolling window," *Journal of monetary Economics,* vol. 41, pp. 455-474, 1998.

[20] S. T. Rachev, S. V. Stoyanov, and F. J. Fabozzi, *Risk and Uncertainty* vol. 211: John Wiley & Sons, 2011.

[21] R. K. Pearson, "Outliers in process modeling and identification," *IEEE Transactions on control systems technology,* vol. 10, pp. 55-63, 2002.

[22] H. Liu, S. Shah, and W. Jiang, "On-line outlier detection and data cleaning," *Computers & chemical engineering,* vol. 28, pp. 1635-1647, 2004.

[23] G. Lawson, M. Sosonkina, T. Ezer, and Y. Shen, "Empirical Mode Decomposition for Modeling of Parallel Applications on Intel Xeon Phi Processors," in *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2017, pp. 1000-1008.

[24] S. J. Pennycook, C. J. Hughes, M. Smelyanskiy, and S. A. Jarvis, "Exploring simd for molecular dynamics, using intel® xeon® processors and intel® xeon phi coprocessors," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, 2013, pp. 1085-1097.

[25] J. Reinders, "An overview of programming for Intel Xeon processors and Intel Xeon Phi coprocessors," *Intel Corporation, Santa Clara,* 2012.