REAL TIME BIG DATA MINING

by

JIMIT PATEL

A thesis submitted to the

Graduate School-Camden

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of Master of Science

Graduate Program in Computer Science

written under the direction of

Dr. Michael A. Palis

and approved by

_____

Dr. Michael A. Palis

_____

Dr. Desmond Lun

_____

Dr. Dawei Hong

_____

Camden, New Jersey

January 2016

THESIS ABSTRACT

Real Time Big Data Mining

By JIMIT PATEL

Thesis Director:
Dr. Michael A. Palis

This thesis presents a parallel implementation of data streaming algorithms for multiple streams. Thousands of data streams are generated in different industries like finance, health, internet, telecommunication, etc. The main problem is to analyze all these streams in real time to find correlation between streams, standard deviation, moving average, etc. There are efficient algorithms available to analyze multiple streams. However, we can still improve the performance of a system to analyze multiple streams through parallel implementation. This thesis specifically focuses on: 1) design and implementation of a parallel system for multiple streams to find Discrete Fourier Transform (DFT), Most Correlated Pair, Singular Value Decomposition, Standard Deviation, Moving Average, and Aggregated Average; 2) performance analysis of multiple threaded application versus single threaded application; and 3) visualization of archived data.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to Dr. Michael Palis, my thesis advisor, for his patience, enthusiasm, encouragement, and great help in research and development. I would like to thank Dr. Andrew Nikiforov, Dr. John Broussard, and Dr. Alok Baveja for motivating me to work harder.

I sincerely thank Dr. Desmond Lun, Dr. Dawei Hong and Dr. Sunil Shende for developing my skills in database system, optimization methods and big data algorithms. I would also like to thank Kevin abbey, system administrator, for providing me a platform to implement and test my system.

I would also like to thank Dr. Ramaswami, a graduate director, for being so much patient and giving me an opportunity to study in this great school.

Finally, I wish to thank my parents, my sister, my brother in law, and my friends for their constant support and encouragement in my study.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Problem statement

Consider the problem of monitoring tens of thousands of time series data streams online and making decisions based on them. In addition to single stream statistics such as average and standard deviation, we also want to find the most highly correlated pairs of streams especially in a sliding window method. A stock market trader might use such a tool to spot arbitrage opportunities. There has been extensive research to find efficient algorithms to calculate these statistics. We discuss all these algorithms. However, these algorithms have not been implemented in parallel. We would like to analyze thousands of real time series in a parallel way so that we can increase performance. To my knowledge, this has not been done. The objective of this work is to parallelize all these methods. Can we actually implement a system using multi-threading? How much improvement we can get in the performance if we use multi-threading? Will there actually be an increase in the performance or will performance drop with the use of multi-threading? These are the questions we want to answer. Hence, we decide to make an application to analyze thousands of real time series in parallel, then compare the performance between single threaded and multi-threaded applications.

1.2 Motivation

There are so many areas where multiple data streams have been used. For example:

1) NASA's Space Shuttle generates data at a rate of 1 sec approximately for 20,000 sensors to Mission Control at Johnson Space Center, Houston [17].
2) Every second up to 10,000 quotes and trades are generated. We want to analyze thousands of trading time series in real time. Traders would like to find correlation, pattern, moving average, standard deviations on the fly. They do not really have enough time to calculate all these values. Things are changing however with the introduction of "Data Stream Management Systems" [18].
3) There are applications in telecommunication. For example, AT&T collects approximately 300 million records per day from its 100 million customers [19].
4) Comcast Cable, a cable company, produces millions of data streams from millions of customer devices. They want to analyze all these data streams in real time because they need to find faults, usage patterns, and unusual activities in progress. This leads to analysis of traffic and fault data in real time [20].
5) This is applied in astronomy. The MACHO project investigates the dark matter in the halo of Milky Way [21]. In this process, they monitor several million stars that generate data at the rate of GBytes per night.
6) Credit card fraud analytics needs to process real time data stream on the fly to predict if a given transaction is fraudulent [15].

7) A wireless sensor network (WSN) is a special kind of adhoc network that has the capacity of sensing and processing, which can be applied to many fields, such as environmental, industrial, military, and agriculture.

Chapter 2 Background and Related Work

More efficient methods for doing analytics over multiple streams based on Discrete Fourier Transform and a three level time interval hierarchy have been discussed in [23]. The algorithm described in [1] beats the direct computation approach by several orders of magnitude. The algorithm is incremental, has fixed response time, and can monitor the pairwise correlations of 10,000 streams on a single PC.

Functional contributions of this thesis are:

1. Computing multi-stream statistics such as synchronous as well as time delayed correlation and vector inner-product in a continuous online fashion. This means that if a statistic holds at time *t*, that statistic will be reported at time *t* + *v*, where *v* is a constant independent of the size and duration of the stream.
2. For any pair of streams, each pair-wise statistic is computed in an incremental fashion and requires constant time per update. This is done using a Discrete Fourier Transform approximation.
3. The approximation has a small error under natural assumptions.
4. Even when the data streams over sliding windows are monitored, no revisiting of the expiring data streams is needed.

In [1], how to compute DFT coefficients incrementally is also discussed. To find out the pairs that are most likely to be correlated, the grid structure discussed in [24] is used in [1]. Then among the filtered pairs, correlation between the streams is found using first n DFT coefficients. Sketches (random projections) which

include different techniques like sketches, convolution, grid structures, structures random vectors, combinatorial design and bootstrapping are discussed in [1]. These techniques have been used to find windowed correlation for uncooperative data streams which do not have concentrated energy over few frequency components. Uncooperative data streams cannot be approximated using data reduction techniques like DFT, Fast Fourier Transform, SVD, Wavelet Transform, etc. Hence, to find correlation between these kinds of data streams with high performance, the sketched approach is used. This approach is discussed in more detail in [26].

One can find great information about models and algorithms for data streams in [14]. The book talks about online algorithms for data stream analysis. Models can be learned from time changing streams without using sliding windows and this is discussed in [25]. In [10], algorithms are developed for interactive visualization of big data. First, scalable visual summaries using data reduction techniques like sampling or binned aggregation were discussed. After that, algorithms for interactive querying like brushing and linking with the use of multivariate data tiles and parallel query processing have been developed. Datar et. al [26] find statistics over a sliding window. An algorithm to monitor statistics over sliding windows at every time point which makes use of a limited memory has been proposed. However, accuracy was compromised. In [27] also, calculating statistics over multiple data streams is discussed. The paper is more focused in calculating correlated aggregates which we are not really interested in finding. The paper presents the summary of correlated aggregates using histogram. For

example, finding the percentage of international calls that last longer than the average value of domestic calls. AT&T has designed a system called Hancock [28] to find signatures from massive data streams. An algorithm to make a decision tree has been proposed in [29] whereas [30] develops an algorithm to make clusters for data streams.

Agrawal et al. [31] transforms the time series into the frequency domain with the help of DFT. Then, the first few DFT coefficients using multidimensional index were indexed. Basic goal was to find whole sequence matching. This technique later on was improved to allow sub sequence matching [32]. In [1], finding similarities among multiple real time series was the main focus. Correlation coefficients were used to find the similarity [33]. There are some other techniques proposed as well to find similarity. For example, [34] discusses about Discrete Wavelet Transform (DWT) and [35] discusses Singular Value Decomposition (SVD). It depends on the dataset as to which technique will work best.

We are mainly focused on proposing an algorithm to use parallelism to find correlation, SVD, standard deviation, DFT, and moving average.

Chapter 3 Summary of Contributions

1. Design and implementation of a system to analyze multiple streams in parallel have been our main contribution in this thesis. We discussed different papers about analysis of multiple data streams. However, analysis of multiple data streams in parallel has not been discussed. That is where we contribute. We make a system that works on parallelizing the analysis of multiple data streams.
2. A parallel radix 2 Decimation in time algorithm to find FFT has been proposed and implemented. Time complexity of the algorithm to find FFT of w points window is O(log w) with O(w) parallel threads giving speed up of w. In real applications, we do not achieve speed up of w. We used this algorithm to find FFT of w points window for n data streams by allocating O(m) threads to n data streams. Thus, we parallelized FFT calculation as well as the calculation for each stream. However, we did not get the speed up in this case due to thread creation overhead. Hence, we propose to use O(m) threads to parallelize calculation for n data streams and single thread for FFT computation. This results in time complexity of O(n/m * w * log w).
3. We use Pearson correlation coefficient to find correlation between streams. We find the most correlated pair among all pairs in n data streams by comparing the Pearson correlation coefficient for each pair. We parallelize this algorithm using O(n) threads and find the most correlated pair from n data streams in O(w * n) time.
4. We find moving average of w points window in O(b) time where b = size of basic window and max of w points window in O(w) time with O(n) threads for n

data streams. We also compute aggregated average of all data streams by taking average of the average value of each data stream. This is done in O(n) time with O(n) threads. We find max of the max value of each data stream in O(n) time with O(n) threads. We show that it is better to use a single thread for moving average as thread creation overhead is greater than computation time. Hence, it is not worth parallelizing the calculation for each stream.

5. We compute standard deviation of w points window in O(w) time. For n data streams, this calculation becomes O(n*w). We parallelize this computation by allocating each thread to each stream.
6. We compute SVD for w points window using LAPACK (Linear Algebra package). This has been calculated in parallel for n data streams with O(n) threads. It is shown that parallelizing SVD calculation for n streams improves performance of the system.
7. We derive in the thesis that we get maximum speed up when we run our program on nodes closely located to each other and when we create threads equal to number of cores. In addition, we conclude that multiple threads only improve the performance of the task when thread creation overhead is negligible compare to actual computation time.
8. We also implement a system to visualize archived data. This system can also be useful to visualize data streams.

Chapter 4 Model and Functions

4.1 Data streams

An important aspect of real time series analysis is the data stream. There are many industries that generate real time data streams. For example, stock exchange data streams, traffic sensor data streams, weather data streams, network data streams, biological data streams, etc. All these data streams share same characteristics like time stamps and data value. For example, for stock market data streams, data is generated at the rate of milliseconds. Every millisecond, a trade happens. So we have time stamp, ticker of the trade, and the value of that ticker.

The thesis intends to analyze the performance of real time data stream analysis. This thesis does not focus on any particular domain. This system is very generic and can be used to analyze any real time data stream. In order to make this system generic, we are synthetically generating the real time data stream. This synthetically generated data stream will possess all necessary properties like time stamps, tag and value.

This system generates data streams similar to the real time series. This data stream has three properties:

1. Time stamp: Each data point that we receive in data stream has a time stamp associated with it. For example, in stock data, we have time when the stock was traded.

2. Tag: This is the tag of generated data. Again, for different kind of data streams, this can be different as well. For stock market data, it is ticker like IBM or Google. For the cable industry, it can be customer id. Some data streams may not have tags. Like traffic censor data, this does not have any tags. These data streams are mainly used to find correlation between streams.
3. Value: This is the actual value of the generated data. For, stock market data streams, it is the value of the trade. For, cable industry, it is the network traffic.

The data stream generation rate can be tuned to our convenience. We can change the rate to measure the performance. Speed can be changed by adding delay.

4.2 Models

We have utilized the data stream model that has been discussed in [1].
The data stream is divided in to three time periods:

- timepoint - the smallest unit of time over which the system collects data, e.g. second.
- basic window - a consecutive subsequence of timepoints over which the system maintains a digest incrementally, e.g., a few minutes.
- sliding window - a user defined consecutive subsequence of basic windows over which the user wants statistics.

The basic window yields three advantages:

1. Results of user queries need not be delayed more than the basic window time.

2. Maintaining stream digests based on the basic window allows the computation of correlations over windows of arbitrary size, not necessarily a multiple of basic window size, as well as time-delayed correlations with high accuracy.

3. A short basic window gives fast response time but fewer time series can then be handled.

To start, the framework subdivides the sliding windows equally into shorter windows, which we call basic windows, in order to facilitate the efficient elimination of old data and the incorporation of new data.

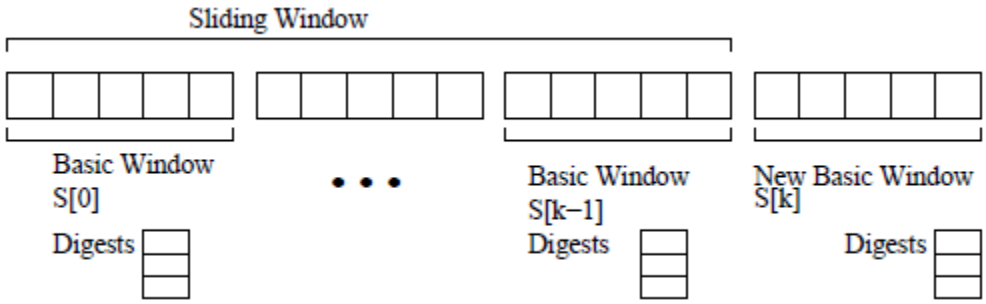


Figure 4.1 Sliding windows and basic windows

Table 4.1 Symbols

w	the size of a sliding window
b	the size of a basic window
k	the number of basic windows within a sliding window
n	the number of data streams

We keep digests for both basic windows and sliding windows. For example, the running sum of the time series values within a basic window and the running sum within an entire sliding window belong to the two kinds of digests respectively. Figure 4.1 shows the relation between sliding windows and basic windows.

Let the data within a sliding window be $s[t - w + 1..t]$. Suppose $w = kb$, where b is the length of a basic window and k is the number of basic windows within a sliding window. Let $S[0], S[1],\ldots,S[k - 1]$ denote a sequence of basic windows, where $S[i] = s[(t - w) + ib + 1..(t - w) + (i + 1)b]$. $S[k]$ will be the new basic window and $S[0]$ is the expiring basic window. The j-th value in the basic window $S[i]$ is $S[i;j]$. Table 4.1 defines the symbols that are used in the rest of the chapter.

4.3 Functions

4.3.1 Moving Average and Max

This function is used to calculate the moving average value of the data stream. To calculate the moving average, we need to maintain sum of the data values which are there in the data stream. Then, we can continuously find the average of the streams by dividing it to number of data points we have seen so far in the data stream. In a sliding window method, we find moving average for the window w = kb. When a new basic window arrives, the sliding window gets updated and we again find the new average of the window. Information to be maintained for the moving average is $\sum(s[t - w + 1..t])$. For each basic window $S[i]$, we maintain the digest $\sum(S[i]) = \sum_{j=1}^{b} S[i; j]$. After b new data points from the stream

become available, we compute the sum over the new basic window *S*[*k*]. The sum over the sliding window is updated as follows:

$$\sum_{new}(s) = \sum_{old}(s) + \sum S[k] - \sum S[0]$$

where $s[t - w + 1..t]$ is a sliding window of w data points, $\sum_{new}(s)$ is the sum over a new sliding window, and $\sum_{old}(s)$ is the sum over an old sliding window.

In terms of run time complexity, we can find the moving average in O(b) time. We maintain the sum over the sliding window. When new basic window arrives, we find sum of data points in the new basic window which takes O(b) time. Updating the sum of the new sliding window takes constant time as we already have the sum of the old sliding window. We just need to remove the sum of the expiring basic window and add the sum of the new basic window. Hence, total time complexity becomes O(b) to find moving average of the data stream.

The max of the window can be found by comparing each value of the window. The max value of each basic window needs to be maintained. When a new basic window arrives, first, max of the new basic window is computed by comparing each data point in the basic window which takes O(b). Then we update max of the sliding window. Updating max of the sliding window is done by comparing max of each basic window. Hence, moving max can be found in O(k*b = w) time.

4.3.2 Moving Average and Max for multiple streams

We find moving averages of multiple data streams using the same sliding window technique. Above, calculating moving average of only one stream is shown. The same way we find moving average for all the streams. We maintain the sum of the sliding window for each data stream. For n data streams, total runtime complexity becomes O(b*n). This can be optimized by using multiple threads. Allocate each thread to find the moving average of each stream. Hence, our time complexity becomes O(b*n/m) provided we have O(m) number. In additional to this, we will get the overhead associated with each thread. Thus, time to find moving average for n streams in parallel can be computed as Tn = Ts/m + mλ where m = number of threads, Ts = time taken by single thread and λ is overhead associated with each thread. Hence, if we do not have large Ts, which is the case here, overhead in creating multiple threads will cause degradation in the performance. We will see this in chapter 6.

For n data streams, max value for each data stream can be computed in O(n*w) time. This can be parallelized allocating each thread to each stream. Hence, it becomes O(n*w/m) with O(m) threads running in parallel.

4.3.3 Aggregated Average and Max

Aggregated max is the max value among max values of the windows of all data streams. Again, this can be found by comparing max values of windows of all data streams. We keep updating sliding windows of all data streams. Hence, we have to find max value again whenever sliding windows get updated. This results

in time complexity of O(n). When we use multiple threads, reduction operation is used to find aggregated max. Hence, time complexity for this becomes O(one reduction operation). One reduction operation is the time to perform reduction operation on critical variable shared by all threads. This depends on the system implementation. Figure 4.2 shows the design of the model. The same way, we can find aggregated average value of the multiple windows of all data streams.

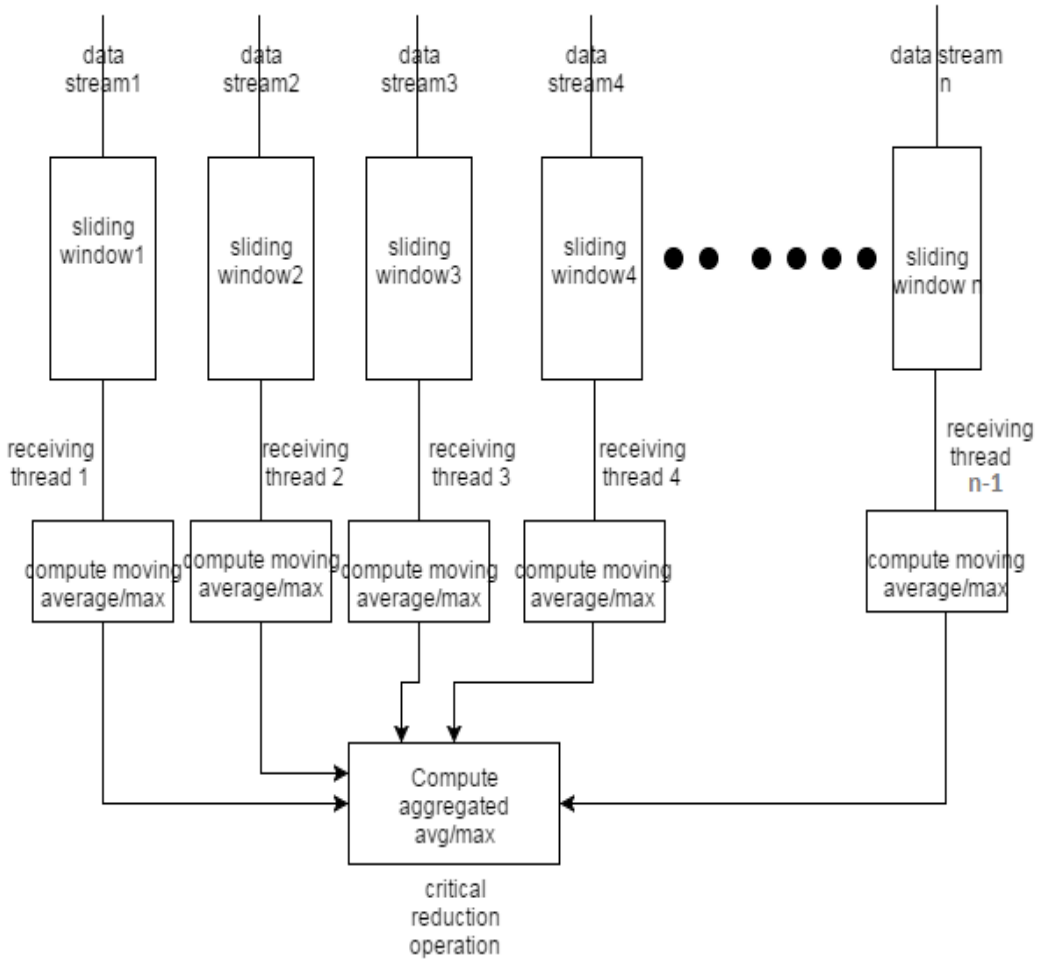


Figure 4.2 Design of computing aggregated average/max

4.3.4 Standard deviation

This function is often used to find how volatile the data stream is. In the case where X takes random values from a finite data set x1, x2, ..., xN, with each value having the same probability, the standard deviation is

$$\sigma = \sqrt{1/N \sum_{i=1}^{N} (x_i - \mu)^2}, where\ \mu = 1/N \sum_{i=1}^{N} x_i \quad (4.1)$$

An application is to find the volatility of the particular stock. Equation 4.1 has been used to find standard deviation. In order to find standard deviation of the sliding window, we need to find average of the window, which is already explained. This average is subtracted from each data point. Finally, we sum up square of these values and divide the final sum by number of data points w. At last, we take the square root of the final value. The Algorithm can be found below,

Find average of sliding window *s*[*t* - *w* + 1..*t*]; //moving average()

Sum =0;

FOR ALL x ∈ sliding window *s:*

Sum= Find summation of (square of (x – average));

Compute square root of (sum/w);

Figure 4.3 Algorithm to find standard deviation

We can see that the time complexity of finding standard deviation for one stream is O(w) where w = sliding window size.

4.3.5 Standard deviation with multiple threads

It can be seen from the algorithm that computation of square (x – average) can be done in parallel across the data points in a sliding window.

create threads { // parallel region starts

FOR ALL x ∈ sliding window s:

t = Compute (square of (x – average));

sum = sum + t // this is a critical update; only one thread should able to do this update at a time

}// parallel region ends

Compute square root of (sum/w);

Figure 4.4 Algorithm to find standard deviation using multiple threads

Given O(m) threads, we can calculate the time complexity of the function = O(w/m + one sum-reduction operation). One reduction operation depends on machine and the way we implement the parallelism.

4.3.6 Standard deviation for multiple streams

For n data streams, standard deviation can be computed using the single thread algorithm discussed in figure 4.3. Run time complexity is O(w * n). We can create multiple threads across n data streams to compute standard deviation. In this case run time complexity becomes O(w * n /m) with O(m) parallel threads.

4.3.7 Discrete Fourier Transform

The discrete Fourier transform (DFT) is a fundamental mathematical operation used in digital signal processing. It allows the user to analyze, modify, and synthesize signals in a digital environment. Because of this, it has found a wide range of uses in engineering and scientific applications. The DFT is performed on a discrete numerical sequence. This is in contrast to the analog Fourier transform, which operates on continuous signals. A discrete sequence is typically a sampling in time of an analog signal, but this is not always the case. For instance, the two-dimensional DFT plays a valuable role in frequency domain image processing. It operates on discrete data representing image pixels, sampled spatially, rather than temporally.

The DFT produces a spectral profile of the frequency components found within a sequence. In other words, it transforms the sequence from a sequence domain (for example, the time domain, or the spatial domain) to the frequency domain. The resulting transformed signal can then be analyzed, or manipulated in ways that are not possible in the sequence domain, or in a manner that would be difficult or time consuming. For example, a common application of the DFT is in digital filtering. If a noisy input is known to contain a useful signal within a known bandwidth, the DFT can be used to first produce a spectral profile of the signal. Next, one can nullify all signal components outside the target bandwidth. When the now modified frequency profile is subsequently transformed back from the frequency domain to its original domain, the undesired noise will be greatly reduced. Though this same operation can be performed outside the frequency

domain, it must be done using time-domain convolution. Convolution becomes prohibitively expensive for anything but small sequences.

The discrete Fourier transform will map a sequence in the time domain into another sequence in the frequency domain. Here is the definition of the Discrete Fourier Transform.

Definition 4.3.1 (Discrete Fourier Transform) *Given a time sequence* $\vec{x}=(x(0),x(1),\ldots,x(N-1))$, *its Discrete Fourier Transform (DFT) is*

$$\vec{x} = DFT(\vec{x}) = \big(X(0),X(1),\ldots,X(N-1)\big)$$

where

$$X(F) = 1/\sqrt{N}\sum_{i=0}^{i=N-1} x(i)e^{-j2\pi Fi/N} \quad F = 0,1,\ldots,N-1 \qquad (4.2)$$

If the time series *X*(*F*) and *x*(*i*) are complex, their real and imaginary parts are real time series:

$$x(i) = xR(i) + jxI(i),$$

$$X(F) = XR(F) + jXI(F),$$

To compute the DFT, we use the following relation:

$$X(F) = 1/\sqrt{N}\sum_{i=0}^{N-1}(x_R + jx_I(i))\left[\cos\left(\frac{2\pi Fi}{N}\right) - isin\left(\frac{2\pi Fi}{N}\right)\right]\ F = 0,1,\ldots,N-1 \quad (4.3)$$

Therefore,

$$X_R(F) = 1/\sqrt{N}\sum_{i=0}^{N-1}[X_R(i)\cos\left(\frac{2\pi Fi}{N}\right) + X_I(i)\sin(2\pi Fi/N)]\ F = 0,1,\ldots,N-1 \quad (4.4)$$

$$X_I(F) = 1/\sqrt{N}\sum_{i=0}^{N-1}[X_I(i)\cos\left(\frac{2\pi Fi}{N}\right) - X_R(i)\sin(2\pi Fi/N)]\ F = 0,1,\ldots,N-1 \quad (4.5)$$

The time series $\vec{x}=(x(0), x(1), \ldots, x(N-1))$ can be thought as samples from a function *f*(*x*) on the interval [0*; T*] such that $x_i = f\left(\frac{i}{NT}\right)$. Although the Discrete Fourier Transform and Inverse Discrete Fourier Transform are defined only over the finite interval [0,*N*-1], in many cases it is convenient to imagine that the time series can be extended outside the interval [0*, N*-1] by repeating the values in [0,*N*-1] periodically. When we compute the DFT based on (4.2), if we extend the variable *F* outside the interval [0*, N*-1], we will actually get a periodical time series. A similar result holds for the IDFT.

4.3.8 Fast Fourier Transform

We will here discuss the efficient algorithm to calculate the DFT which is FFT. Here, we will discuss about the radix 2 Decimation in Time (DIT) algorithm to find FFT. The decimation in time (DIT) radix-2 FFT is the most intuitive FFT algorithm, and the simplest to derive, so it will be presented first. It is also the same algorithm presented in the original Cooley and Tukey paper [36] on the FFT. The term *decimation in time* refers to the method of derivation. Given the DFT for a discrete data sequence in time,

$$X[m] = \sum_{t=0}^{N-1} x[t] W_N{}^{mn} \tag{4.6}$$

where N is the length of x[t] and WN = $e^{-j2\pi/N}$. WN is known as the (1/N)th root twiddle factor. The DIT FFT follows by recursively splitting the DFT of x[t] into multiple, smaller DFTs of subsequences of x[t]; in other words, to decimate x[t] in time. For the radix-2 DIT FFT, x[t] will be recursively decimated into two smaller sequences of length N/2. Given the discrete sequence x[n] = {x[0], x[1], ..., x[N

−1]}, where N is power of- two, the DFT of x[n], X[m], is given by (4.6). The summation of (4.6) can be split into two summations of length N/2,

$$X[m] = \sum_{n=0}^{N/2-1} x[2n] W_N^{m2n} + \sum_{n=0}^{N/2-1} x[2n+1] W_N^{m(2n+1)}$$

$$= \sum_{n=0}^{N/2-1} x[2n] W_{N/2}^{mn} + W_N^{m} \sum_{n=0}^{N/2-1} x[2n+1] W_{N/2}^{mn}, \quad (4.7)$$

Where the identity $W_N^2 = W_{N/2}$ is used. Now observe that the upper-half of $X[m]$ can be obtained from the bottom half, giving

$$X[m+N/2] = \sum_{n=0}^{N/2-1} x[2n] W_{N/2}^{(m+N/2)n} + W_N^{(m+N/2)} \sum_{n=0}^{N/2-1} x[2n+1] W_{N/2}^{n(m+N/2)}$$

$$= \sum_{n=0}^{N/2-1} x[2n] W_{N/2}^{mn} - W_N^{m} \sum_{n=0}^{N/2-1} x[2n+1] W_{N/2}^{nm} \quad (4.8)$$

This holds because

$$W_N^{n(m+N/2)} = W_{N/2}^{mn} W_{N/2}^{nN/2} = W_{N/2}^{nm}, \quad (4.9)$$

and

$$W_N^{m+N/2} = W_N^{m} W_N^{N/2} = -W_N^{m} \quad (4.10)$$

By comparing equations (4.9) and (4.10), it can be seen that x[m] and x[m+N/2], for 0<=m<N/2, only differ by a sign. Therefore, the two halves of the DFT result can be produced by using the same operands. These operands are the two summations in (4.9) and (4.10); they are two *N*/2-point DFTs. Because the input sequence, $x[n]$, has been recursively decimated in time, the input order has been scrambled. For FFTs of a power of two radix, the data can be reordered by a simple *bit-reverse-copy*. This consists of copying the input sequence into a new sequence where the elements have been assigned to bit-reversed addresses. After the bit-reverse copy, the data can be presented to the radix-2 DIT FFT in the correct order. Despite the need to reorder the data, the computational savings of the FFT are considerable, compared to the DFT.

We have described pseudocode of the radix-2 DIT FFT of the iterative algorithm. Though the DIT FFT was derived recursively, recursive algorithms are difficult to map to hardware. Note that the outer loop iterates log(N) times where N=w=size of the window, and the inner loop iterates N times. Also, lines 11-14 perform the twiddle factor multiplication and the radix-2 butterfly. Line 4 completes a bit-reverse copy of the input data sequence.

```
1. ITERATIVE_DIF_FFT(x,X) {
2.      X = x; /* copy x */
3.      n = length(x);
4.      bit_reverse(X); /*reorder X */
5.      for(s = log(n); s >= 1; s--) { /* outer loop */
6.              m = 2^s;
7.              wm = cos(2*pi/m)-sqrt(-1)*sin(2*pi/m);
8.              for(k = 0; k < n; k += m) { /* inner loop */
9.                      w = 1; /* twiddle factor */
10.                     for(j = 0; j < m/2; j++) { /* execute butterflies */
11.                             t = X[k+j];
12.                             u = w*X[k+j+m/2];
13.                             X[k+j] = t + u;
14.                             X[k+j+m/2] = t - u;
15.                             w = w*wm; /* compute next twiddle factor */
16.                     }
17.             }
```

18. }

19.}

The pseudocode for a recursive version can be found below.

1. Recursive-FFT (array)
2. - arrayEven = even indexed elements of array
3. - arrayOdd = odd indexed elements of array
4. - Recursive-FFT (arrayEven)
5. - Recursive-FFT (arrayOdd)
6. - Combine results using Cooley-Tukey butterfly
7. - Bit reversal, could be done either before, after or in between

The time complexity of the sequential FFT algorithm is O(w log w) where w = size of the window.

4.3.9 Parallel Fast Fourier Transform

Figure 4.5 shows the graph in the case of 16 data points. The first column of butterflies connects grid points whose numbers differ in bit 0 (the leftmost bit). For example, the black butterfly connects 0000 and 1000. The second column of butterflies connects grid points whose numbers differ in bit 1; for example the black butterfly connects 0000 and 0100. The next black butterfly connect 0000 and 0010, and the rightmost black butterfly connects 0000 and 0001. Each of the rightmost columns can be computed in one time step using 16 processors.

Psudo code for parallel version of iterative fft:

Input: n = number of data points, m = $\log_2$ n, A_re = real part of input data, A_im = imaginary part of input data

Output: n points DFT stored in A_re and A_im for real and imaginary coefficients respectively

```
fft(n,m,A_re, A_im)
{
Bitreversal(A_re,A_im);       /*bit reversal *
n1=0;   /*FFT*/
n2=1;
FOR(i=0;i<m;i++)    /* for each stage out of total log(n) stages*/
{
        n1=n2;
        n2=2*n2;
```

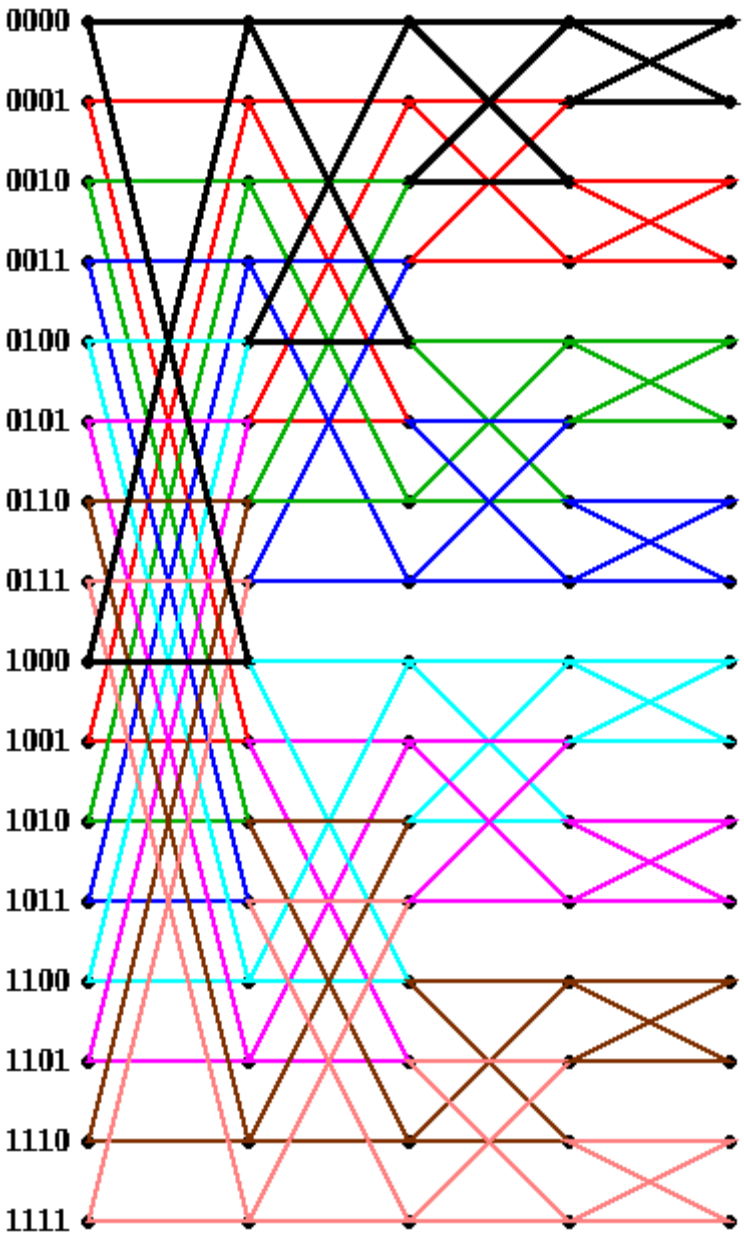


Figure 4.5 Data dependencies in a 16 points FFT

e=-6.2381853071/n2;

a=0.0;

create parallel threads{

```
        FOR (j=0;j<n1;j++){  /* each butterfly group uses only one root of unity.*/
          a=j*e;
          c=cos(a);
          s=sin(a);
         FOR(k=j;k<n;k=k+n2){  /* execute each butterflies */
            t1=c*A_re[k+n1] - s*A_im[k+n1];
            t2=s*A_re[k+n1] + c*A_im[k+n1];
            A_re[k+n1]=A_re[k]-t1;
            A_im[k+n1]=A_im[k]-t2;
            A_re[k]=A_re[k]+t1;
            A_im[k]=A_im[k]+t2;
        }
      }
    } // end of parallel region
}
}
```

Psudo code for recursive version of parallel fft:

```
pcfft(input)
{
  IF (n <= WINDOW) /*if n satisfies this condition then perform sequential fft*/
    {
    cfft(n,wn,X,Y); // call to a sequential FFT function
    RETURN;
    }
m = n/2;
FOR (j = 0; j < m; j++)
  {
```

```
    X1[j] = X[2*j];
    X2[j] = X[2*j+1];
  }
 # CREATE PARALLEL TASK1
{
 pcfft(input1);
}
# CREATE PARALLEL TASK2
{
 pcfft(input2);
}
# wait for task 1 and task 2 to complete
w.r = 1.0;
w.i = 0.0;
FOR (j=0;j<m;j++)  /*Combine results using Cooley-Tukey butterfly */
  {
  Y[j] = Cadd(Y1[j],Cmult(w,Y2[j]));
  Y[j+m] = Cdif(Y1[j],Cmult(w,Y2[j]));
  w = Cmult(w,wn);
  }
}
```

Described parallel algorithm runs in O(w/m log w) time for w size sliding window with O(m) parallel threads. Each time sliding window is updated, DFT for the new sliding window is computed.

4.3.10 Fast Fourier Transform for multiple streams

In a scenario of n multiple streams, we compute DFT using parallel iterative FFT algorithm for each stream. Run time complexity can be calculated as O (w/m * n/m * log w) with O (m) parallel threads for data streams and O (m) threads in FFT function.

4.3.11 Pearson Correlation Coefficients

This is used to find the correlation between two time series. The value of pearson coefficients varies between -1 to 1. -1 means negatively correlated and 1 means positively correlated [3]. This is used in almost every industry to find the relationship between two streams. If two streams are highly correlated then we can predict the future value of the other stream based on the value of the one stream. Finding most correlated pair takes time because we have to compare each and every pair to find correlation coefficient. We use the pearson correlation coefficient to find the correlation between two streams. The algorithm can be found in [3]. Here's the formula for computing the coefficient for two items x and y

$$pearson(x,y) = \frac{\sum(xy) - \frac{(\sum x)(\sum y)}{n}}{\sqrt{(\sum(x^2) - \frac{(\sum x)^2}{n})(\sum(y^2) - \frac{(\sum y)^2}{n})}}$$

4.3.12 Monitor Most Correlated Pair

Pearson coefficient has been used to find correlation between streams. The pseudocode to find the most correlated pair among n streams can written as,

FOR each x € [1, n] // for each data stream

FOR each i € [x+1, n] // compute pearson coefficient with

FOR each data points in a window

SUM of x; SUM of j

SUM of the squares of x; SUM of the squares of j

SUM of the PRODUCT of (x,j)

COMPUTE pearson score

COMPARE pearson scoring for each j and UPDATE local most correlated

COMPARE local most correlated pair for each x and UPDATE global most

correlated pair

In order to find most correlated pair, coefficients between each pair in multiple data streams are calculated. If we have n data streams, then we need to make total n(n-1)/2 comparison. Hence this algorithm is O(w * n * n).

We can use multi-threaded application to parallelize the function. Multiple threads will handle the outer ***for*** loop in parallel. Each x for data steam has been allocated a separated thread to compute local most correlated pair in parallel. We can reduce the time complexity to O(w * n/m) by using O(m) threads to process n data streams. Here, there is a critical section to update the global most correlated pair across all the data streams. This update is done sequentially by each thread resulting in O(m) complexity. Thus the time complexity will be O(w * n/m + m) with

O(m) threads. However, we do not have balance load for each thread to work as the comparison starts reducing with high

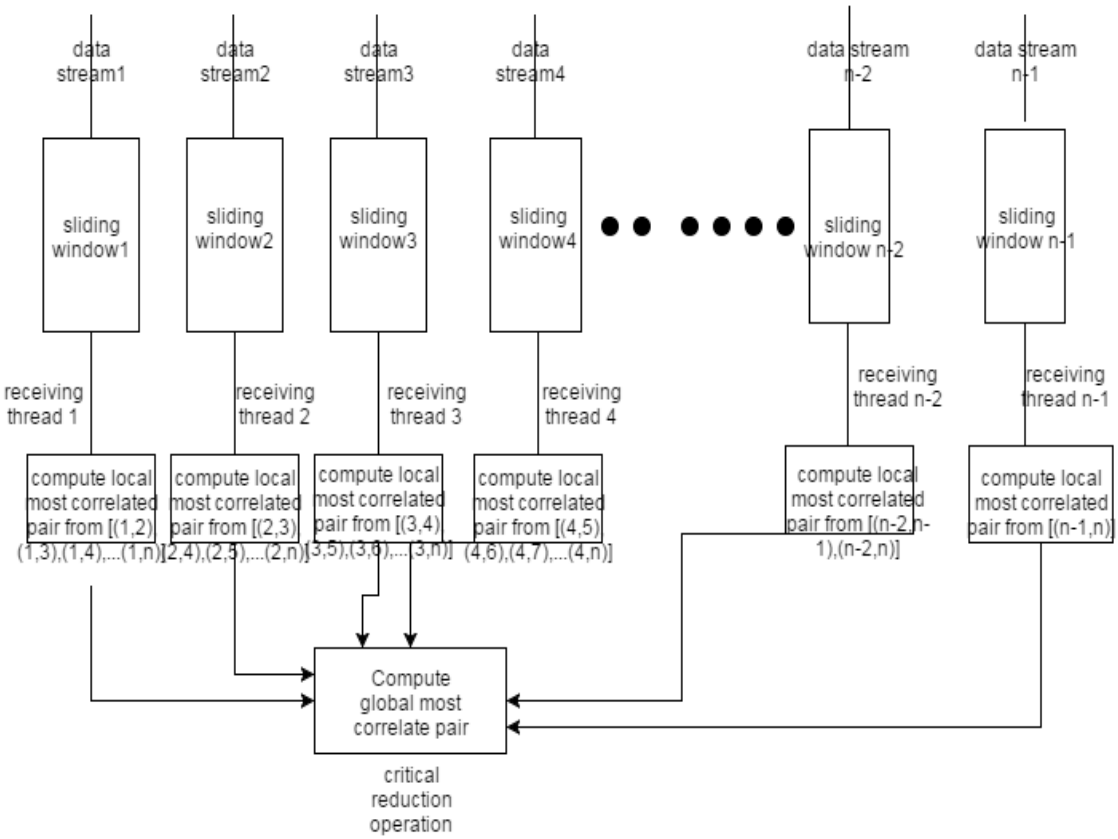


Figure 4.6 Design for computing most correlated pair

value of x. For example, for x=1, we have to compare 1 with all the streams starts from 2 to n. But, in the case of x=n-2, the comparison is only from n-1 to n. Thus, we need to distribute our load same to each threads for better performance. This can be implemented in OpenMP with scheduled(dynamic,1) command. This basically distributes the load among the threads dynamically. When one thread finishes the task early, it will be allocated the remaining iteration to compute. When a new basic window comes, sliding window gets updated and we compute most

correlated pair again across all the data streams. Figure 4.6 shows design of the model.

4.3.13 Singular Value Decomposition

The Discrete Fourier Transform and the Discrete Wavelet Transform of different wavelet families are all based on the orthogonal transform of time series. We choose the orthogonal basis vectors based on the nature of the time series and keep a few coefficients of the transform to approximate the original time series. One might wonder whether there is an optimal orthogonal transform that is the best in approximating the original time series with as few coefficients as possible. The answer is positive. It is the Singular Value Decomposition.

First we introduce the concept of *Singular Value Decomposition*. For more detail about SVD, the readers can refer to [22]. Consider a collection of *m* time series $\vec{x}_i, i = 1,2,\dots,m,$ each time series has the same length *n*: $\vec{x}_i, i = (x_i(1), x_i(2), \dots, x_i(n))$. We can write such a collection of time series in the following matrix form:

$$\mathbf{A}_{m\times n} = \begin{pmatrix} \vec{x}_1 \\ \vec{x}_2 \\ \vdots \\ \vec{x}_m \end{pmatrix} = \begin{pmatrix} x_1(1) & x_1(2) & \dots & x_1(n) \\ x_2(1) & x_2(2) & \dots & x_2(n) \\ \vdots & \vdots & \ddots & \vdots \\ x_m(1) & x_m(2) & \dots & x_m(n) \end{pmatrix}. \tag{4.10}$$

We can write for any symmetric matrix S = A A^T,

$$S\ \overrightarrow{U_i} = \lambda_i\ \overrightarrow{U_i}\ ,\ i = 1,2,\dots,d \tag{4.11}$$

Where $\overrightarrow{U_i} = eigenvectors, i = 1,2,\dots,d$ and $\lambda_i = eigenvalues\ , i = 1,2,\dots,d$

Let the rank of s be d. Of course, d<= m. If S is singular, then

$$\lambda_{d+1} = \lambda_{d+2} = \lambda_m = 0$$

Using linear algebra we can also define

$$\overrightarrow{V_i} = 1/\sqrt{\lambda_i}\ \ A^T\overrightarrow{U_i}\ , i = 1,2,\dots,d. \tag{4.12}$$

$\{\overrightarrow{U_i}\}_{i=1,2,\dots,d}$ and $\{\overrightarrow{V_i}\}_{i=1,2,\dots,d}$ are orthonormal vectors.

Hence, $A = I_{m*m}\ A$

$$= \sum_{i=1}^{m} \overrightarrow{U_i}\ \overrightarrow{U_i^T}\ A$$

$$= \sum_{i=1}^{m} \overrightarrow{U_i}\ (A^T\ \overrightarrow{U_i})^T$$

$$= \sum_{i=1}^{d} \sqrt{\lambda_i}\ \overrightarrow{U_i}\overrightarrow{V_i}\ + \sum_{i=d+1}^{m} \overrightarrow{U_i}\ (A^T\ \overrightarrow{U_i})^T$$

We can prove using simple linear algebra $\sum_{i=d+1}^{m} \overrightarrow{U_i}\ (A^T\ \overrightarrow{U_i})^T = 0$

Then we can write

$$A = \sum_{i=1}^{d} \sqrt{\lambda_i}\ \overrightarrow{U_i}\overrightarrow{V_i} \tag{4.13}$$

Equation (4.13) is called the *Singular Value Decomposition (SVD)* of the matrix A. $\sqrt{\lambda_i}, i = 1,2,\dots,d$ are the singular values of A. The vectors $\{\overrightarrow{U_i}\}_{i=1,2,\dots,d}$ and $\{\overrightarrow{V_i}\}_{i=1,2,\dots,d}$ are the left singular vectors and right singular vectors of the decomposition respectively.

SVD can be found solving for eigenvalues and eigenvectors of A . A^T and A^T . A. The efficient and more practical algorithm to find SVD is a Jacobi algorithm

proposed by James Demmel and Krešimir Veselic [38]. The algorithm implicitly computes the product AA^T and then, uses a sequence of "Jacobi rotations" to diagonalize it. A Jacobi rotation is a 2 × 2 matrix rotation that annihilates the off-diagonal term of a symmetric 2 × 2 matrix. Given a 2 × 2 matrix M with

$$M^T M = \begin{bmatrix} \alpha & \gamma \\ \gamma & \beta \end{bmatrix}$$

It is possible to find a ``rotation by angle θ matrix''

$$\Theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

with the property that $\Theta^T M^T M \Theta = D$ is a diagonal matrix. Θ can be found by multiplying out the matrix product, expressing the off-diagonal matrix component in terms of $t = \tan\theta$, and setting it to zero to arrive at an equation

$$t^2 + 2\zeta t - 1 = 0$$

where $\zeta = (\beta - \alpha)/(2\gamma)$. The quadratic formula gives $t = \tan\theta$ and $\sin\theta$ and $\cos\theta$ can be recovered. There is no need to recover θ itself. The algorithm repeatedly passes through the implicitly-constructed matrix AA^T , choosing pairs of indices i < j, constructing the 2 × 2 submatrix from the intersection of rows and columns, and using Jacobi rotations to annihilate the off-diagonal term. Unfortunately, the Jacobi rotation for the pair i=2, j =10 messes up the rotation from i=1, j=10. So, the process must be

repeated until convergence. At convergence, the matrix Σ of the SVD has been implicitly generated, and the right and left singular vectors are recovered by multiplying all the Jacobi rotations together.

Chapter 5 System Implementation

5.1 Computation of Time Series

The computation of the time series is done in a receiver part. A producer continuously produces the data streams at the rate which we can specify in the system. After producing the data, it stores it into the buffer. The receiver thread will take the data out from the buffer and processes it to find different analytical values. The high level architecture of the model can be seen in the figure 5.1.

The system is basically a two thread application where one thread is the producer and the other thread is the receiver. If we specify multiple streams, then the producer will produce the data for multiple streams and store it into the respective buffer. The receiver removes data from the buffer and does the analysis we want to perform. If there are multiple streams, then the receiver can be multi-threaded. Each thread is allocated to perform analysis for one stream. Thus, we can have parallel implementation of the system. This receiver is again the multi-threaded application. We want to compare the performance between the multi-threaded receiver and the single threaded receiver. When we have multiple data streams and the single threaded receiver, then the receiver has to handle all the streams one by one. This takes time if we have large number of data streams coming in. Instead, if we have multiple receiver for each data stream, then each receiver will handle each data stream. This computation is done in parallel. We can reduce our time for analysis by implementing parallel system.

The computation also depends on the configuration of the system. Multi-threaded application will be only useful if we have multi-core system. We can allocate multiple cores to different threads and they all can work in parallel. If we do not have multiple cores then multiple threads will have to wait for their turn to access the core. Eventually, this will reduce the performance because of the thread creation overhead. We will discuss more about this in chapter 6. Figure 5.1 describes the high level description of the system. The producer produces multiple data streams into the buffer for each data stream. Then, the receiver will create multiple threads to handle each data streams. Each thread first takes the data out from the buffer and puts it into sliding window. Then, each receiver computes output like DFT, Moving average, standard deviation, SVD, local most correlated pair, etc. Then, we can calculate global most correlated pair, aggregated moving average, aggregated max value, etc. Finally, we can feed output to the visualization system like matlab to visualize the aggregated output. Figure 5.1 describes the high level architecture of the system. Now we will study more detailed implementation of the system. A First important part of the system is the buffer.

5.2 Design of Buffer

A Buffer is basically the storage for the data streams. The producer thread produces the data and stores it into the buffer. This buffer is accessible to both: the producer and the receiver. In the system implementation, we have used a two dimension array to store the data. One dimension is for multiple stream and second is for buffer size. The producer will put the data into the buffer till it becomes full.

Once the buffer is full, it will stop producing data till there is a space to put the data. We can vary size of the buffer. In the system, we have used struct window r[world_size -1][BufferSize] as the buffer. Structure window has time and value properties. World_size is the parameter that defines the number of streams and BufferSize is the parameter that defines the size of the buffer.

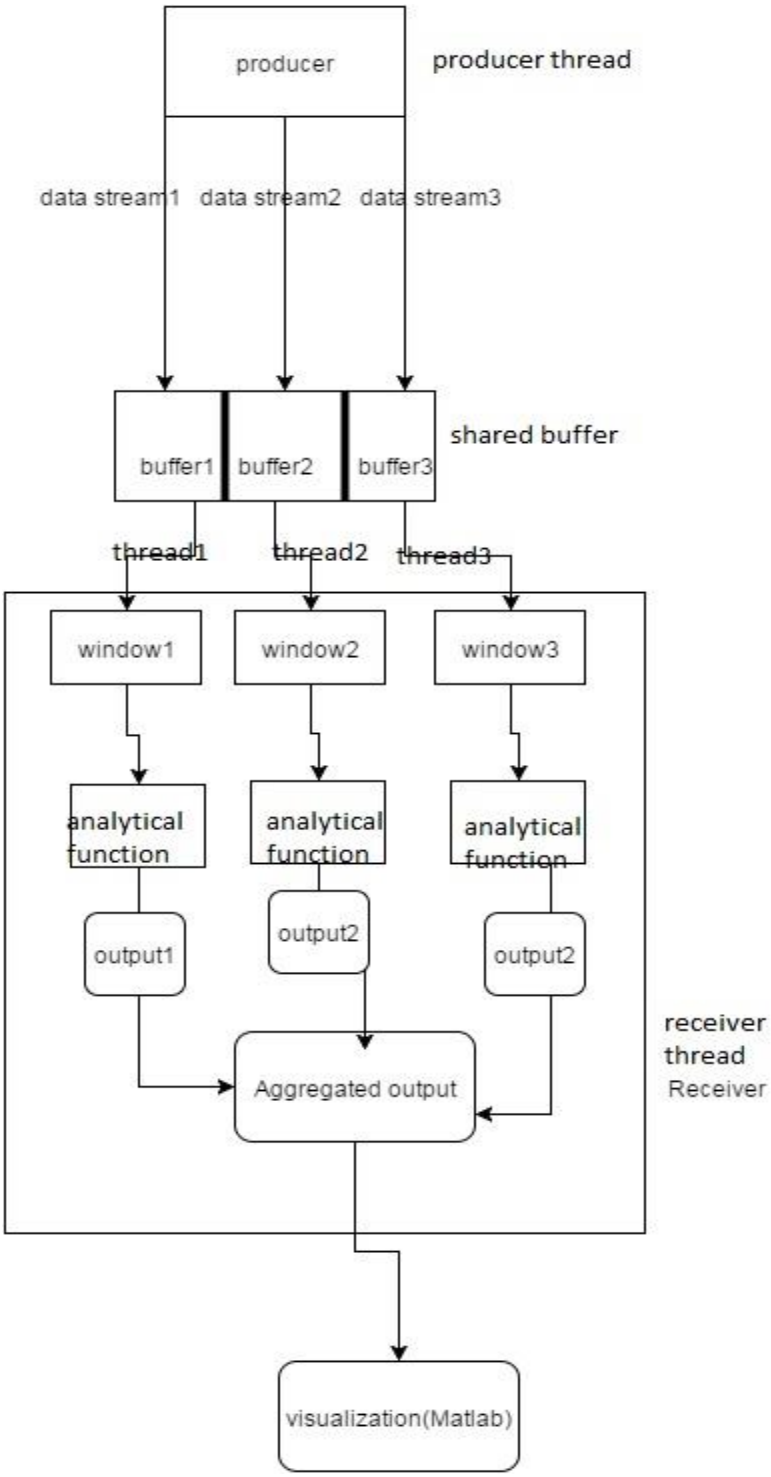


Figure 5.1 High level description of the system

This means that each stream has space = Buffer size to store the data and each data has two fields: time and value.

5.3 Design of Producer

A producer is the thread which produces data continuously. The producer uses the random generator to produce the random number. However, we need to make sure the random generator follows Gaussian distribution. For that, we have implemented a function which follows normal distribution. The function has been implemented based on the Gaussian distribution definition. This random generator will also ensure that it will generate different range of data for different data streams. The function will generate normal distribution data around the value x. The value of x depends on the rank of the data stream. For example, for stream number 0, it will generate normal distribution data centered on value 70; for stream number 1, it will generate normal distribution data centered on value 140.

For multiple data streams, the producer will keep producing data for each stream one after one in sequential manner. For that, we have used ***for*** loop that will iterate for each data stream. This loop is sequential and can be parallelized to speed up producing speed. If producer is faster than receiver, then producer might fill up the buffer. In this case, it will have to wait for the receiver to release some space from buffer to put the data into it. This producer can be set to feed live data from different data sources. It can be set to finance application, feed data from censor, etc.

5.4 Design of Receiver

A receiver is the thread that consumes the data from buffer and processes it to find different analytical functions. The receiver takes the data from buffer and this process can be implemented in parallel. Each thread can be allocated to each stream to process the data. We use the sliding window technique discussed in the chapter 4. It continuously updates the sliding window when a new basic window arrives. Then, it finds different statistics on the sliding window. Size of the basic window and sliding window can be changed for different rates of producer and receiver.

```
1. For(;;) //This is for the receiver which keeps receiving data from buffer
   continuously
2. {
3.     Update start time;
4.     For (all i <= number of streams)// This is to take data for each stream and
                                        can be parallelized
5.     {
6.         Update window;  // this updates the sliding window based on receiving
                             index
7.         Compute different analytical values;
8.     }
9.   Update receiving index // increment receiving index after receiving data
10.  Update end time;
11.  Compute consumed time// finds total time taken in computation
```

12. CHECK IF buffer is empty: // it checks if buffer is empty or not
13. Wait on TRUE // will wait for producer to produce
14. }

A basic design of the receiver has been shown in pseudo code above. We can see that the loop runs for each data stream. For each data stream, we update the sliding window and calculate different statistics. In this system, if the receiver is faster than the producer, then it will have to wait for producer to generate the data in to the shared buffer. We start our timer to compute the performance when the receiver is able to fetch data from the buffer. We end our timer when we get the analytical output for all data streams. This way we make sure that our time consumption does not include waiting time. We try to avoid waiting time for the receiver by using large buffer size. We also start the producer little bit early for the first time. So that by the time the receiver starts processing the data, the buffer is already full. Moreover, except moving average function, all other analytical functions take more time to process the data for all the streams than the producer to produce it.

Based on models discussed in chapter 4, analytical functions have been implemented. We discussed algorithm to find standard deviation for n data stream in parallel in figure 4.3. Run time complexity is O(w * n/m) with O(m) parallel threads for n data streams. The performance of the same has been discussed in chapter 6. However, we do not parallelize the computation within function discussed in figure 4.4 as thread overhead kills the performance.

Out of the two parallel FFT versions, the iterative algorithm is faster than the recursive version. One of the main reasons recursive version is not faster that it becomes very difficult to map threads to the task. The overhead associated in creating the task every time the recursive call to function is made, is higher. We will discuss more about the performance of parallel iterative FFT algorithm in chapter 6. We already discussed in section 4.2.9 that run time complexity of the algorithm is O(w/m log w) for O(m) threads. In the case with n data streams, run time becomes O(w/m * n/m * log w) with O(m) parallel threads for data streams and O(m) threads in FFT function. The performance is discussed in chapter 6.

5.5 Open MP Introduction

- **OpenMP Is:**
  - An Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism
  - Comprised of three primary API components:
    - Compiler Directives
    - Runtime Library Routines
    - Environment Variables

- **Shared Memory Model:**
  - OpenMP is designed for multi-processor/core, shared memory machines.

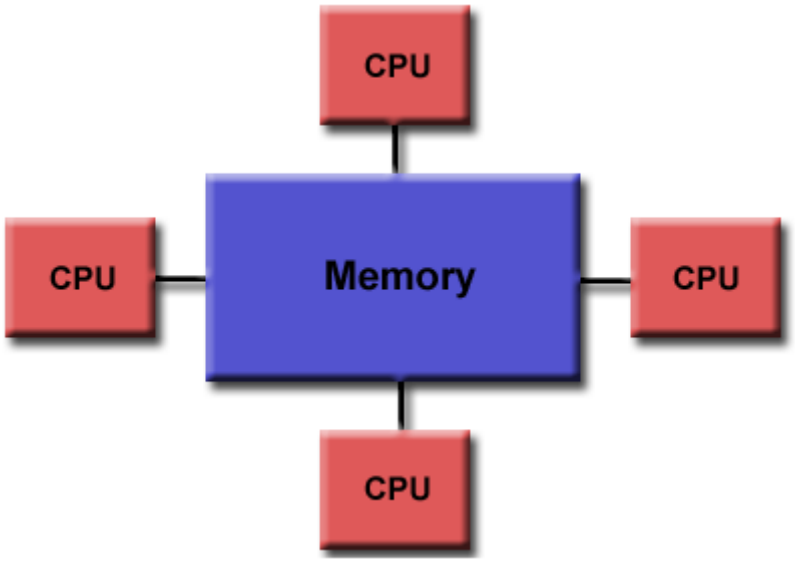


Figure 5.2 Uniform memory access

- **Parallelism:**
  - OpenMP programs accomplish parallelism exclusively through the use of threads.
  - Typically, the number of threads matches the number of machine processors/cores. However, the actual use of threads is up to the application.
  - OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization.
  - Parallelization can be as simple as taking a serial program and inserting compiler directives or as complex as inserting subroutines to set multiple levels of parallelism, locks and even nested locks.

- **Fork - Join Model:**

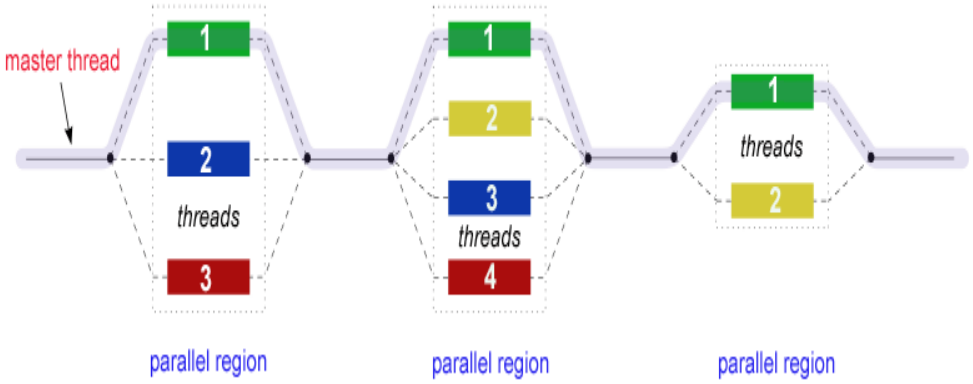


Figure 5.3 Fork-Join model

- OpenMP uses the fork-join model of parallel execution.
- All OpenMP programs begin as a single process: the **master thread**. The master thread executes sequentially until the first parallel region construct is encountered.
- **FORK**: the master thread then creates a team of parallel threads.
- The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads.
- **JOIN**: When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.

5.6 Parallel implementation

In order to achieve the parallelism, we utilize the OpenMP. First, we need to create two threads: one for the producer and another for the receiver. This can be done using openMP. To use OpenMP functionality, we need to add <omp.h> header in our C program.

#pragma omp parallel num_threads (2)

The above command will create two threads in the program. OpenMP is shared memory architecture. This means when you create two threads within a program, each thread will share the same memory. If we want to make any variables private to any particular thread, we can specify it using ***private*** keyword while creating threads. More details can be found in the document [5]. Using omp_get_thread_num(), we can find the thread number. Then, we can allocate producer to one thread and receiver to another thread. Thus, both producer and receiver will work in parallel.

Inside the receiver thread, we need to create parallel threads to process multiple data streams in parallel. For that, we need to enable nested openMP. Nested openMP is used to create threads within threads. ***omp_set_nested(1)*** will enable nested openMP.

#pragma omp parallel for reduction (max:aggr_max) reduction(+:aggr_avg)

This command is in the receiver. This will create multiple threads to process ***for loop***. Each thread will handle each iteration in parallel. ***reduction*** keyword is used to perform reduction operation on the variable that is shared by all the threads. This variable should only be accessed by one thread at a time to avoid race condition.

reduction (max:aggr_max) will find max value of aggr_max variable that is being updated by each thread. The same way we can find sum using ***+.***

We have also used ***#pragma omp critical*** functionality. This will make the following section critical. Hence, only one thread can enter to this section at a time. Once the one thread finishes the work, another thread will enter the section and start the task. This way we can take care of the race condition and can find the most co-related pair among all the streams.

In the case of calculating most correlated pair, to distribute the load evenly among all the threads we use ***scheduled (dynamic,1).*** This will basically do the load balancing dynamically and can solve the problem of uneven distribution of load in each iteration. There are other ways of doing explicit load balancing. For example, we can use n(n-1)/2 threads and map these threads to n(n-1)/2 pairs. We can also use n(n-1) threads. However, we did not carry out these approaches in this thesis.

Chapter 6 Testing and Results

This chapter discusses the results that we get in our system. We will compare the performance of single threaded application and multi-threaded application. We will discuss which application is better for which function. We will also discuss why one application is performing better than the other application. The performance is calculated based on speed of the receiver to process different analytical functions. We will find the start time before the start of function. Then, we will find the end time once the function ends. We can calculate the time taken by taking the difference between start time and end time. Time taken is measured in micro seconds.

We have used a vsmp cluster that has total 8 nodes with each node having 8 cores. The detailed architecture of the vsmp cluster can be found here. According to the vsmp architecture, all the nodes from 0 to 7 are connected to each other through infinity band. However, we can say that the pair of nodes 0-1, 2-3, 4-5, 6-7 are the closely located pair to each other. Hence, if we run our program on these pair of nodes, we get the optimal performance. If we use any different combination of nodes, then the performance of the system will decrease. For example, if we use node 0,3 to run our openMP program, then the path that cluster takes to reach out to node 3 is: 0-1-2-3. Hence, we can see that it will route from 0 to 1, 1 to 2, and then 2 to 3. This will take some extra time in terms of reaching to the designated node in the cluster. Thus, whatever performance we got with node 0-1, we will definitely get less than that in the case of node 0-3. Hence, we should understand our cluster configuration to better understand the

performance of the model. This is the reason why we use cores mapped to the closely located pair or single node.

6.1 Performance Analysis for Discrete Fourier Transform

For the different input length n and for different number of threads, we have measured the time taken in microseconds to calculate the FFT. The program was run on cores 0-15 from nodes 0-1. We used export GOMP_CPU_AFFINITY=0-15 command to tell the system to use cores 0-15. This is how we control the thread distribution in openMP to analyze the performance.

Figure 6.1 presents the time taken for calculating FFT for different length of input sequences with different number of threads. Fig 6.2 shows us the speed us for different number of threads for w = 65536 = window size. The result of the plot clearly shows that the maximum performance we get is at thread number 16. The clear difference in time is visible when we have higher length of input sequence like w=65536. For higher w, threads get more computation and there will be higher speed up. Remember, all these tests have been performed by selected cores 0-15 from node 0,1. We can see that we get maximum performance at threads = 16 because we have total 16 cores available. Increasing number of threads to 16 and 24 adds additional waiting time to get the resources. Thus, it reduces the performance. The same pattern is visible when we select cores 0-7. With 8 cores, we get the maximum performance at threads =8 and the performance degrades as we further increase the number of threads. Hence, we can say that maximum performance we get is when we create same number of threads as number of available cores.

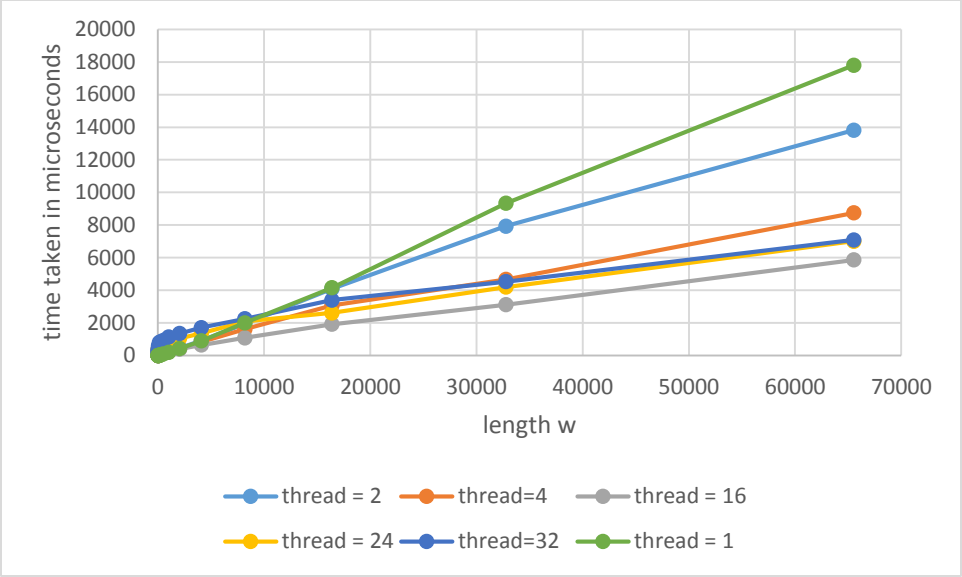


Figure 6.1 Input length n versus time

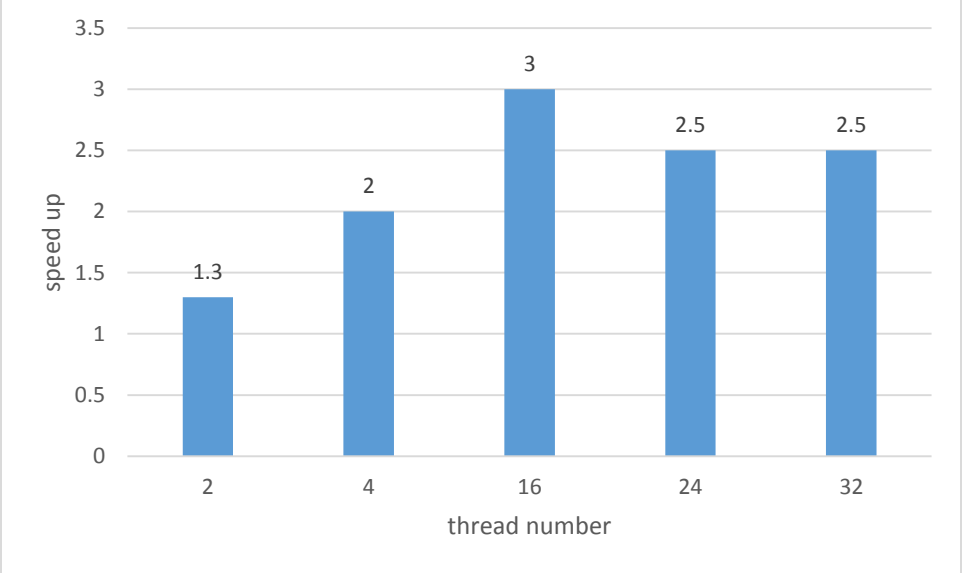
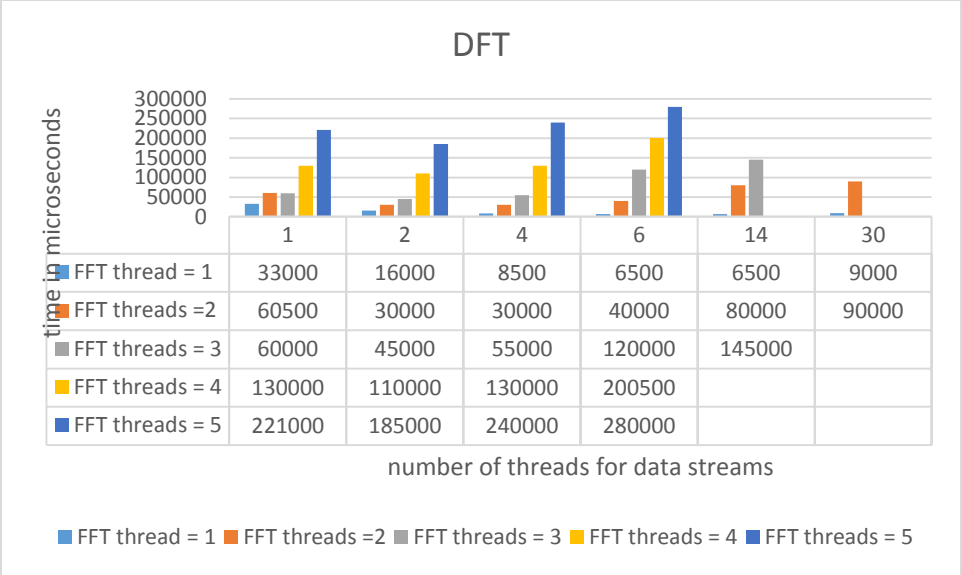


Figure 6.2 Thread number vs speedup



	1	2	4	6	14	30
FFT thread = 1	33000	16000	8500	6500	6500	9000
FFT threads =2	60500	30000	30000	40000	80000	90000
FFT threads = 3	60000	45000	55000	120000	145000	
FFT threads = 4	130000	110000	130000	200500		
FFT threads = 5	221000	185000	240000	280000		

Figure 6.3 Number of threads for data streams vs time

Figure 6.3 shows the performance of calculating DFT using multiple threads for data streams and multiple threads in FFT function. The program had streams = 40, buffer size = 4096, and window = 4096. This performance was measured by submitting job to one node. The graph shows the performance when parallel threads within FFT function and for multiple data streams have been created to calculate DFT for 40 data streams. It can be seen that for thread=1 in FFT function, we have the optimum performance when we use 6 threads to compute DFT of 40 data streams. When we tried to parallelize FFT function also, the performance started degrading. For two threads in FFT function, the optimum performance to compute DFT of 40 streams was found when we created 4 parallel threads to handle 40 streams. The total time was 5.6 times higher than what we got in the case of thread = 1 in FFT function. The same way performance kept on degrading when we tried incrementing parallel threads in FFT function. This clearly tells that parallel threads in FFT function kills the performance in our system. In parallel

iterative FFT algorithm for w point window, we have parallel *for* loop for butterfly group of each log (w) stage. For each iteration of outer *for* loop of log (w) stage, parallel *for* loop of butterfly group has to create multiple threads. OpenMP creates threads only once in this case. Then, it uses those threads for each outer loop iteration. However, when we use this algorithm in our system that uses nested OpenMP, it creates new threads for each outer iteration. This kills the performance of the system.

Figure 6.4 shows the performance for calculating DFT of 40 streams by creating parallel threads to handle 40 streams. This experiment was performed by keeping single thread in FFT function. The program had streams = 40, buffer size = 4096, and window = 4096. The job was submitted to two nodes as well as to a single node. X axis shows the number of threads and Y axis represents the time take in micro seconds to complete the task. We can say from the graph that for nodes=2 and thread=1, it takes 33,000 µs. The same task using 14 number of threads takes 4500 µs. This means that we get the speed up of 7.33 if we run the program using 14 threads. This was really expecting because calculating DFT for each stream using single thread in sequential manner is a slower process. Hence, we can improve our performance using multiple threads. The result also shows another thing that we can't create as many threads as we want. We need to find the ideal number for threads that gives the optimal performance. We can see this number is 14 as the total cores available are 16. If we create more threads than the number of available cores, then threads will have to wait for the cores to finish the task. Moreover, OpenMP also needs to run scheduler to allocate cores to many

threads. There is an overhead associated with the scheduler to allocate resources to the threads. We can also change the default configuration of the scheduler. More detail about scheduler can be found in [6]. Thus, creating more threads than the available cores will decrease the performance.

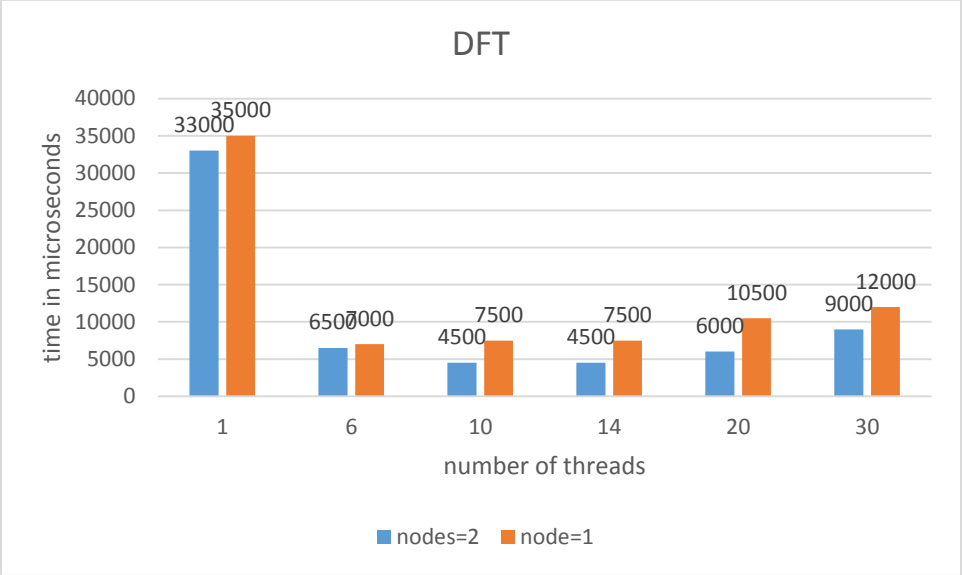


Figure 6.4 Number of threads vs time for calculating DFT

The result also displays that submitting job to two nodes gives better performance than single nodes. It is because two nodes gives more number of cores to handle multiple threads. For node=1, best performance was found at number of threads = 6. It took 7000 µs to compute DFT for 40 streams. Again, for two nodes, it was optimal at threads = 14 and time was 4500 µs.

6.2 Performance Analysis of Correlation

Correlation function shows the same trend as the DFT. The program had streams=200, buffer size = 1000, and window size = 1000. Single thread application took 30000µs to find the most correlated pair of the window. The same

task run with 16 threads took 9000µs. There was a speed up of 3 in the performance with 16 threads. It is shown in figure 6.5. If we use two nodes with threads 30, the same task is done in 7000 µs. This is 3000 µs lesser than one node. The performance is shown in figure 6.6.

6.3 Performance Analysis of Standard Deviation, Moving Average and Max

This program had with streams = 1000, buffer = 1000, window = 200.

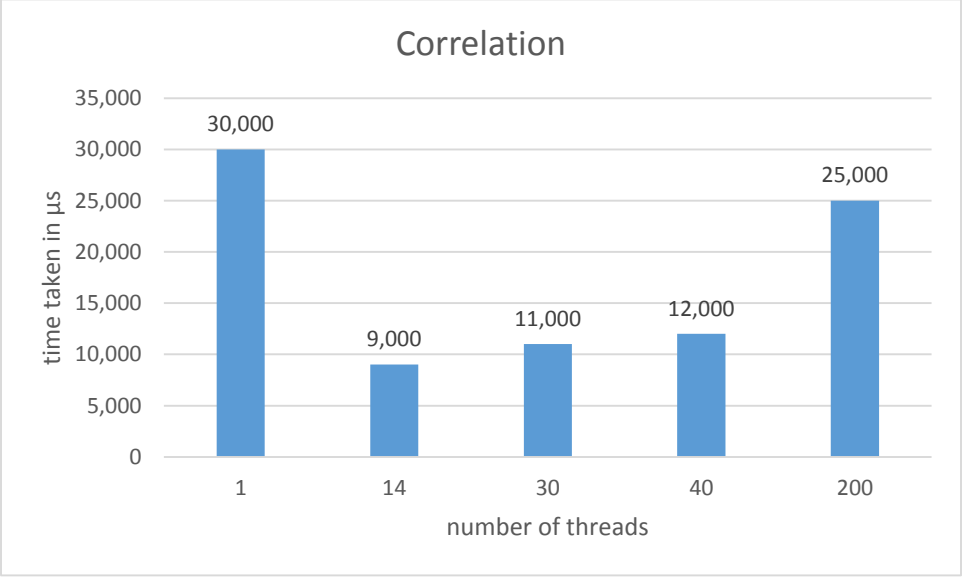


Figure 6.5 Number of threads vs time for calculating correlation

The program was run on a single node with 8 cores. Figure 6.7 shows performance of standard deviation. If we run the task using single thread, it takes around 3100 µs. if we use threads = 6 then, it takes around 800 µs. The performance is optimum at number of threads =6 as we had 8 cores available. The speed up we get is 3.75. We can also see that the performance starts degrading when we further increase number of threads. Figure 6.8 shows performance for calculating max. It also follows the same pattern and gives maximum performance at number of threads =

6. Interesting result was found in performance of calculating moving average. It is optimum for thread = 1. It degrades rapidly when we increase number of threads. This is because of thread creation

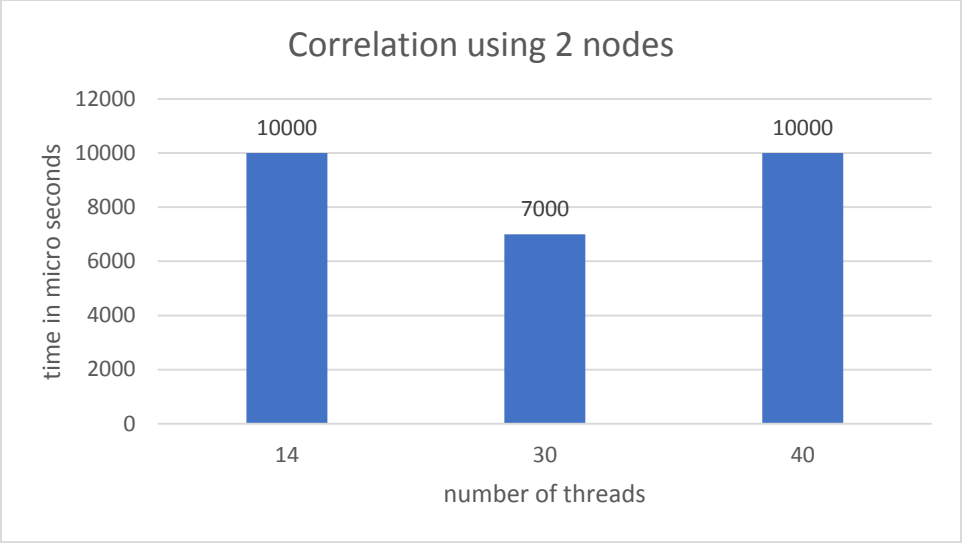


Figure 6.6 number of threads vs time for calculating correlation using 2 nodes

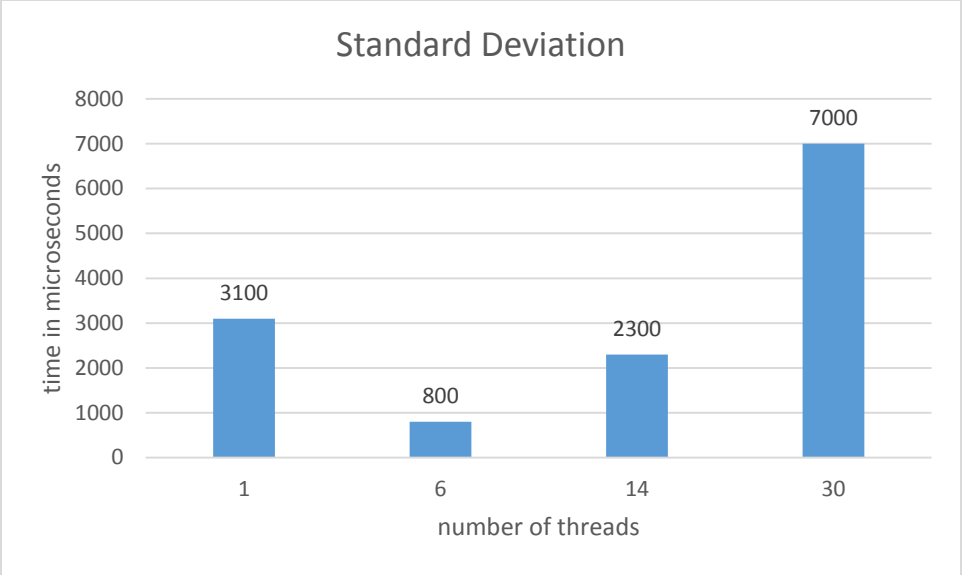


Figure 6.7 Number of threads vs time for standard deviation

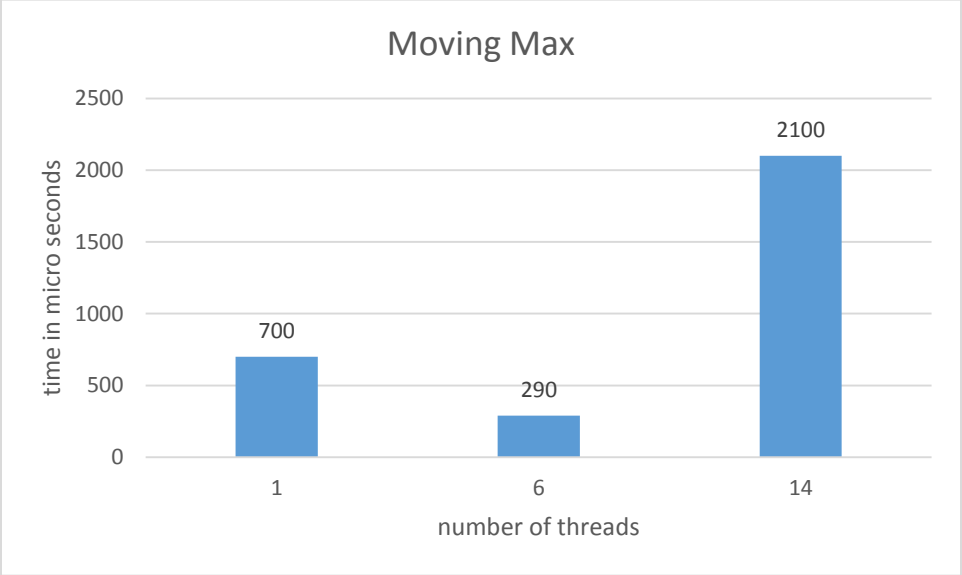


Figure 6.8 Number of threads vs time for moving average

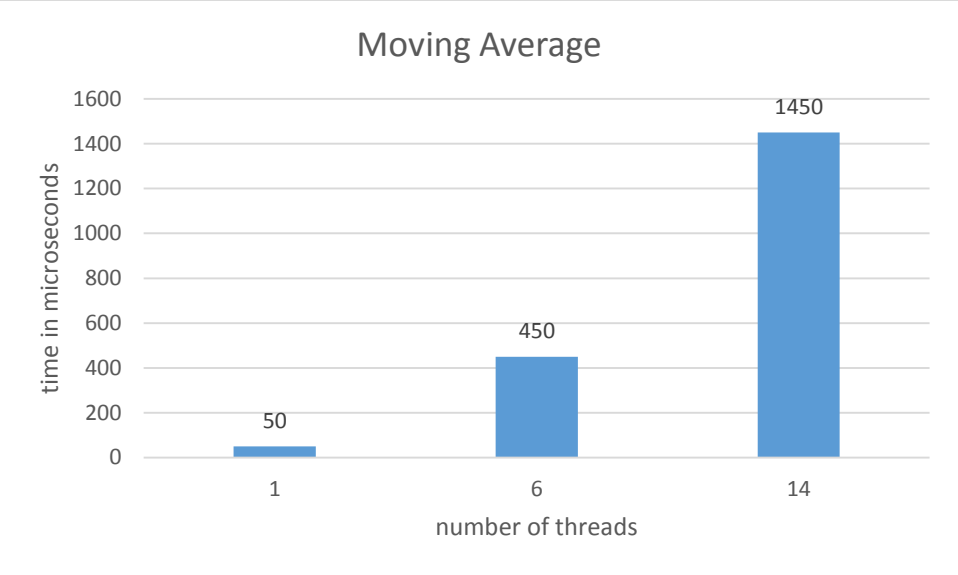


Figure 6.9 Number of threads vs time for moving average

overhead. Single thread itself takes only 50 µs to execute this task. Now, if we create more threads to execute this task, then there will also be an overhead associated with it. More detail about the overhead can be found in intel OpenMP performance tuning paper [7]. There are total four types of overhead associated with the thread creation. 1. Thread library startup overhead 2) Thread startup overhead 3) Per-thread overhead 4) Lock management overhead. This is the main reason behind the graph that we get for moving average. Theoretically, we should be able to increase performance. However, thread creation overhead supersedes the actual time taken to calculate the function. Hence, it does not make sense to create more threads.

6.4 Performance Analysis of Singular Value Decomposition:

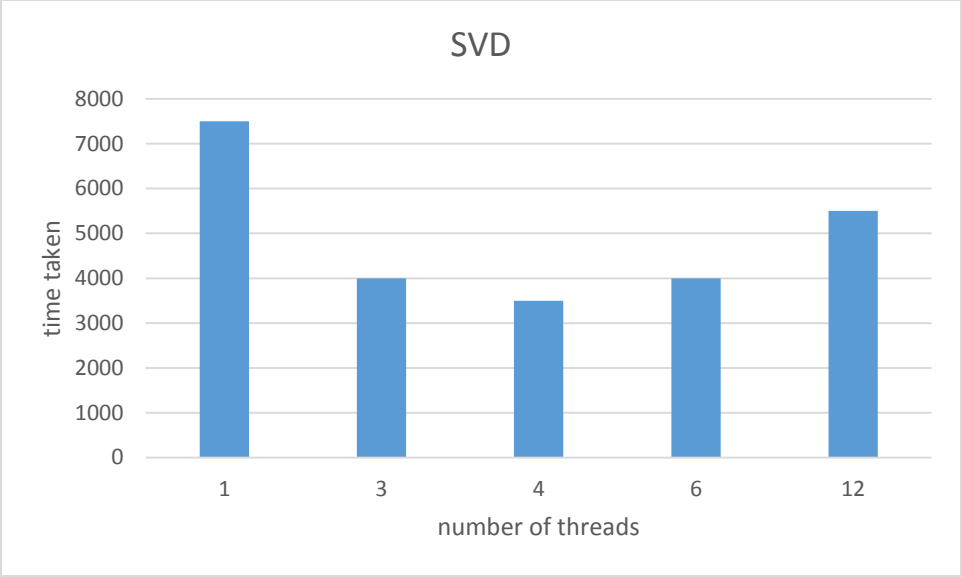


Figure 6.10 Number of threads vs time for SVD

The graph in figure 6.9 shows the performance for computing SVD. The program had streams = 200 and window size = 100. It was run on a cluster with cores = 8. The result shows us that the maximum speedup we get is when we have number of threads = 4.

Chapter 7 Visualization

7.1 Introduction

This chapter is focused on implementing a program to visualize the data stream. There is a need of a system which can visualize the data coming at a speed of milliseconds. We have proposed an implementation in matlab which can visualize archived data or data stream. This code can also be integrated with the c code that we implemented to analyze multiple data streams. Integrating matlab call with c code gives ability to visualize the analytical output of the C program.

7.2 Data sample

We had a Nanex actual trading data in archived form [13]. The sample of the data can be seen below,

```
;{Header}=
LstExg|Sym|SsDte|SsID|NxTime|ExgTime|RepExg|MMID|
;{OptTail}= ;

TD|{Header}|Trade|Tick|Size|ExgSeq|VolType|Open|High|Low|Last|NetCh|TickVol|TotVol|{
Optio nTail} ; RB|{Header}|Bid|BidCh|BidSz|BidSzCh|{OptionTail} ;
RA|{Header}|Ask|AskCh|AskSz|AskSzCh|{OptionTail} ;
BB|{Header}|BBid|BBidCh|BBidSz|BBidSzCh|{OptionTail} ;
BA|{Header}|BAsk|BAskCh|BAskSz|BAskSzCh|{OptionTail} ;
MB|{Header}|Bid|BidCh|BidSz|BidSzCh|{OptionTail} ;
MA|{Header}|Ask|AskCh|AskSz|AskSzCh|MMQteType|MMType|

RB|NYSE|eNKE |04/05/2010|0|09:30:00.000|09:29:59|PACF| | 72.97| -0.01| 1| -4|
RA|NYSE|eNKE |04/05/2010|0|09:30:00.000|09:29:59|PACF| | 74.54| 0.00| 3| 0|
BB|NYSE|eNKE |04/05/2010|0|09:30:00.000|09:29:59|PACF| | 74.08| 0.00| 3| 0|
BA|NYSE|eNKE |04/05/2010|0|09:30:00.000|09:29:59|PACF| | 74.54| 0.00| 3| 0|
RB|NYSE|eNKE |04/05/2010|0|09:30:00.000|09:29:59|PACF| | 72.96| -0.01| 4| +3|
RA|NYSE|eNKE |04/05/2010|0|09:30:00.000|09:29:59|PACF| | 74.54| 0.00| 3| 0|
BB|NYSE|eNKE |04/05/2010|0|09:30:00.000|09:29:59|PACF| | 74.08| 0.00| 3| 0|
BA|NYSE|eNKE |04/05/2010|0|09:30:00.000|09:29:59|PACF| | 74.54| 0.00| 3| 0|
RB|NYSE|eNKE |04/05/2010|0|09:30:00.000|09:29:59|PACF| | 72.97| +0.01| 1| -3|
```

RA|NYSE|eNKE |04/05/2010|0|09:30:00.000|09:29:59|PACF| | 74.54| 0.00| 3| 0|
BB|NYSE|eNKE |04/05/2010|0|09:30:00.000|09:29:59|PACF| | 74.08| 0.00| 3| 0|
BA|NYSE|eNKE |04/05/2010|0|09:30:00.000|09:29:59|PACF| | 74.54| 0.00| 3| 0|

BB (Best Bid price): It is the maximum price that a trader can get from the investors.

BA (Best Ask price): It is the minimum price that a trader can get from the seller.

7.3 Implementation

We use Matlab for visualization. Matlab provides tool especially for the real time data visualization. Timer object is the one which is used in visualization. Datafeed toolbox is the tool which is specifically designed for the trading visualization.

Timer: This is an object which is used in real time visualization. The object has the following parameters as inputs:

- TimerFcn: This is a call back function which is called each time timer fires.
- ExecutionMode: This is the mode of execution. Here, it is set to 'fixedRate'.
- Period: The period at which the timer fires. It can be set to minimum 0.001 sec.

We first need to take data from the archived file and then visualize the data in plot. We need to update the plot once the new data come in. Timer object in matlab fires at the time period specified in the object. When the object fires, it calls the callback function defined in the code. If timer period is set to 1 sec, then it continuously calls callback function at 1 sec. Then, the call back function fetches data from file line by line in each call by timer function. It also plots the output in

continuous form. Call back function can contain basic logic. We can check for the particular symbol match. If the particular ticker matches, take the data. Otherwise, discard it. We can put basic analysis like price is higher or lesser than the previous time stamp. We have put graphs for three different prices BB, BA, and TD for the ticker.

7.4 Result

Below, the demo of what we did with visualizing trading data at 1 second interval can be seen. The result here is shown for ticker ‘eCSCO’. It shows the graphs for three prices BB, BA, and TD. These graphs will give the traders a good reasoning to analyze the results.

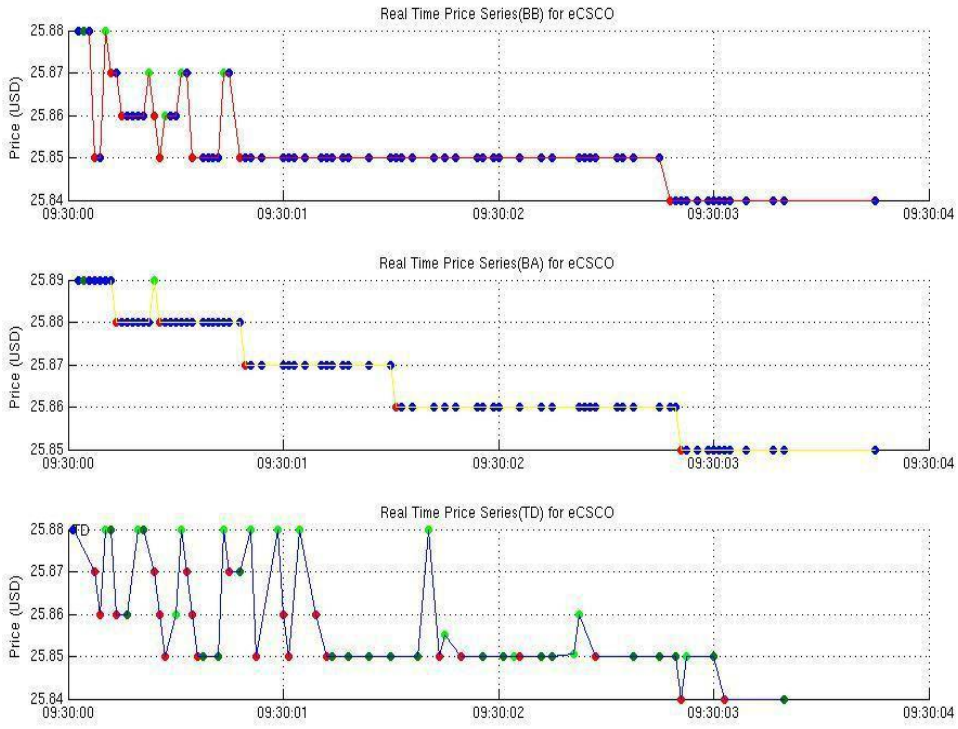


Figure 7.1 Visualization of Cisco trading data

Chapter 8 Future Work

Although, we have got the performance increase in this thesis, there are still some recommendations for future work. We have implemented functions to calculate DFT, standard deviations, most correlated pair, Moving average, etc. However, we have not worked on creating incremental algorithms for these functions. We basically update our sliding window for each new basic window. Then, we calculate all the functions again. We do not find incremental max or incremental standard deviation for the sliding window. A book, Data Streams Models and Algorithms by Charu C. Aggarwal [8] describes an incremental algorithm to find max. We can find incremental algorithm for standard deviation in [9].

We have used brute-force approach to find the most correlated pair among the multiple streams. This can be further optimized using the approach given in paper [1]. We can use indexing and hashing to filter out pairs which have Euclidean distance less than a threshold value based on the first few coefficients of the DFT. Then, we can calculate most correlated pair from the remaining pairs using DFT.

Another work in this system we can do is that we can include Matlab for visualization. We have already done visualization using Matlab for real time stock market series. We visualize stock values in real time at the interval of 1 second. We can use this code to visualize the different analytics values that we find from our system. We need to call Matlab engine from C program [11,12].

Chapter 9 Conclusion

Three conclusion can be drawn from this thesis. First, parallel implementation in the system definitely helps improving the performance in a significant way. Calculating DFT and most correlated pair using multiple threads speed up the performance of the system. DFT and most correlated pair functions have lots of work to be done by a single thread if we work on many streams. In this case, if we parallelize this work by creating multiple threads for each stream, we can significantly improve the system performance. We can apply this in the industry for real time series analysis.

Second conclusion we can draw is that number of threads, which is equal to available cores, gives the optimal performance. It is because maximum number of threads, which can access the cores at a time, is number of available cores. The remaining threads have to wait for their turn to access the cores. This will take more time and reduce the performance. We can increase available cores by running the program on multiple nodes. However, this will have the overhead of distributing threads to multiple nodes.

Third conclusion we can draw from our results is that parallel implementation is only useful for functions which have heavy computation. For function like moving average, it is not good idea to work in parallel unless we have so many streams like 10000 or window size like 1000 to handle. If single thread does not have heavy calculation, then creating multiple threads for the task will have overhead that supersedes the time taken by single thread. Hence, we should create multiple threads only when there is a lot of computation for single thread.

Appendix A Testing Routine

1. **ssh to vsmp cluster.**
2. **Set the buffer size, size of the window, and number of streams.**
3. **gettimeofday(&start,NULL);**
4. **First set the number of omp threads =1**

   omp_set_num_threads(1);
5. **program to calculate Moving Average.**
6. **gettimeofday(&finish,NULL);**

   (double)(finish.tv_usec - start.tv_usec));
7. **Run the program using,**

   numactl --cpunodebind=0 ./prod-consu4_2_4 > output.log
8. **See the output file for the time in micro seconds to finish moving average job**
9. **Now, set**

   omp_set_num_threads(14);

   Rerun the program to measure performance using 14 threads.

We can measure the performance for DFT, SVD, Standard Deviation, Most correlated pair, Aggregated Moving Average the same way**.**

References

[1] Yunyue Zhu. *High Performance Data Mining in Time Series: Techniques and Case Studies* (Dissertation). New York University, January 2004.

[2] Normal Distribution. In Wikipedia. Retrieved from https://en.wikipedia.org/wiki/Normal_distribution

[3] Pearson Correlation Coefficient. In Wikipedia. Retrieved from https://en.wikipedia.org/wiki/Pearson_product-moment_correlation_coefficient

[4] Goertzel Algorithm. Wikipedia. https://en.wikipedia.org/wiki/Goertzel_algorithm

[5] Tim Mattson, Larry Meadows. *A Hands on Introduction to OpenMP.* In OpenMP. Retrieved from http://openmp.org/mp-documents/omp-hands-on-SC08.pdf

[6] OpenMP. Wikipedia. Retrieved from https://en.wikipedia.org/wiki/OpenMP

[7] Paul Lindberg. *Performance Obstacles for Threading: How do they affect OpenMP code?* Intel, January 2009. Retrieved from https://software.intel.com/en-us/articles/performance-obstacles-for-threading-how-do-they-affect-openmp-code

[8] Charu C. Aggarwal. Data Streams Models and Algorithms. *In Springer Science, 2007.*

[9] Calculate standard deviation: case of sliding window. *MATLABtricks.com, October 2013*. Retrieved from http://matlabtricks.com/post-20/calculate-standard-deviation-case-of-sliding-windo

[10] Zhicheng Liu, Biye Jiangz, and Jeffrey Heer. imMens: Real-time Visual Querying of Big Data. In *proceedings of the Eurographics Conference on Visualization (EuroVis)*, Volume 32 (2013), Number 3.

[11] Call MATLAB Functions from C/C++ Applications. Mathworks. Retrieved from http://www.mathworks.com/help/matlab/matlab_external/calling-matlab-software-from-a-c-application.html

[12] Introducing Matlab Engine. Mathworks. http://www.mathworks.com/help/matlab/matlab_external/introducing-matlab-engine.html

[13] NxCore Historical Data. Nanex. http://www.nanex.net/historical.html

[14] S. Muthukrishnan. Data Streams: Algorithms and Applications.

[15] Dibyendu Bhattachary and Manidipa Mitra. Analytics on big fast data using real time stream data processing architecture.

[16] Haixun Wang, Jian Pei, and Philip S. Yu. *Online Mining of Data Streams: Problems, Applications and Progress*. IBM T.J. Watson Research Center, USA; Simon Fraser University, Canada. Retrieved from https://www.cs.sfu.ca/~jpei/publications/seminar4-online-mining.pdf

[17] E. Keogh and P. Smyth. A probabilistic approach to fast pattern matching in time series databases. In *the third conference on Knowledge Discovery in Databases and Data Mining*, 1997.

[18] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3):109-120, 2001.

[19] C. Cortes, K. Fisher, D. Pregibon, and A. Rogers. Hancock: a language for extracting signatures from data streams. In *ACM SIGKDD Intl. Conf. on Knowledge Discoveryand Data Mining*, pages 9-17, 2000.

[20] D. Barbara, W. DuMouchel, C. Faloutsos, P. J. Haas, J. M. Hellerstein, Y. E. Ioannidis, H. V. Jagadish, T. Johnson, R. T. Ng, V. Poosala, K. A. Ross, and K. C. Sevcik. The new jersey data reduction report. *Data Engineering Bulletin*, 20(4):3-45, 1997.

[21] http://www.macho.mcmaster.ca/project/overview/status.html.

[22] W. Press, B. Flannery, S. Teukolsky, and W. Vetterling. *Numerical recipes: The art of scientific computing*. Cambridge University Press, 1986.

[23] Arthur Whitney and Janet Lustgarten. *The high-performance database that sets the standard for time-series analytics, 1993.* Retrieved from http://www.kx.com.

[24] J. L. Bentley, B. W. Weide, and A. C. Yao. Optimal expected-time algorithms for closest point problems. *ACM Transactions on Mathematical Software (TOMS)*, 6(4):563-580, 1980.

[25] G. Hulten, L. Spencer, and P. Domingos. Mining time-changing data streams. *In Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 97-106. ACM Press, 2001.

[26] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *In Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, January 6-8, 2002, San Francisco, CA, USA. ACM/SIAM, 2002, pages 635-644, 2002.

[27] J. Gehrke, F. Korn, and D. Srivastava. On computing correlated aggregates over continual data streams. In ACM SIGMOD *International Conf. on Management of Data*, 2001.

[28] C. Cortes, K. Fisher, D. Pregibon, and A. Rogers. Hancock: a language for extracting signatures from data streams. In ACM SIGKDD *Intl. Conf. on Knowledge Discovery and Data Mining*, pages 9-17, 2000.

[29] P. Domingos and G. Hulten. Mining high-speed data streams. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 71-80. ACM Press, 2000.

[30] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams. In *the Annual Symposium on Foundations of Computer Science, IEEE*, 2000.

[31] R. Agrawal, C. Faloutsos, and A. N. Swami. Efficient Similarity Search In Sequence Databases. In D. Lomet, editor, *Proceedings of the 4th International Conference of Foundations of Data Organization and Algorithms (FODO)*, pages 69-84, Chicago, Illinois, 1993. Springer Verlag.

[32] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *Proc. ACM SIGMOD International Conf. on Management of Data*, pages 419-429, 1994.

[33] C.-S. Li, P. S. Yu, and V. Castelli. Hierarchyscan: A hierarchical similarity search algorithm for databases of long sequences. In *ICDE*, pages 546-553,1996.

[34] K.-P. Chan and A. W.-C. Fu. Efficient time series matching by wavelets. In *Proceedings of the 15th International Conference on Data Engineering, Sydney, Australia*, pages 126-133, 1999.

[35] F. Korn, H. V. Jagadish, and C. Faloutsos. E±ciently supporting ad hoc queries in large datasets of time sequences. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 289-300, 1997.

[36] J. W. Cooley and J. W. Tukey. An algorithm for the mssachine calculation of complex fourier series. *Mathematicl Computations*, 19, April 1965.

[37] Joseph McRae Palmer. *The Hybrid Architecture Parallel Fast Fourier Transform* (Thesis). Brigham Young University, 2005

[38] James Demmel and Krešimir Veselic. *Jacobi's method is more accurate than QR,* pages 32-38. Retrieved from http://www.netlib.org/lapack/lawnspdf/lawn15.pdf