High Frequency Parallel Crypto Asset Valuation

Riccardo Santoni
School of Computer Science
Carnegie Mellon University
Pittsburgh, USA
rsantoni@andrew.cmu.edu

Zongpeng Yu
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, USA
zongpeny@andrew.cmu.edu

SUMMARY

The world of high frequency trading is a type of algorithmic trading characterized by high speeds. The use of sophisticated algorithms, specialized hardware, and co-location is prevalent in the field, enabling competitors to parse market data, price the asset, and immediately send an order back out in the fraction of a millisecond. Our specific application focuses on the first two challenges of the problem space in the crypto markets: parsing market data and pricing an asset by aggregating a vast number of market data to produce our own internal view of the market. Specifically, by using a 2.4 GHz 8-Core Intel Core i9 CPU, we were able to achieve a 7.75x sustained throughput across a total of 480 asset/exchange pairs(12 exchanges and 40 assets).

1 BACKGROUND

1.1 High Frequency Trading Overview

In general, the world of high frequency trading is a cutthroat environment, where competitors are racing each other at nano/millisecond speed scales to exploit market opportunities. There are a number of key software components for live trading. To begin with, there are market data parsers, normalizing each exchange's protocol (including different versions over time) into the same internal format. Then there is the actual strategy component, receiving the normalized market data and acting on it. Oftentimes proprietary trading firms first run an internal valuation model on the data to price the asset, and then pass on the data to an actual trading strategy. Finally, if a decision to trade has been made, this code would communicate back with the order gateway software, converting the internal order format to match each exchange's order entry protocol. This entire flow can happen in the fraction of a microsecond (30 microseconds via a general purposed processor. 300 nanoseconds via an FPGA), and further at a high sustained throughput, aided by the use of specialized hardware, extreme optimizations, and abundant parallelization [1].

1.2 Project Scope

For our project specifically, we focused on the first two aspects of live trading; parsing market data from a number of exchanges, allowing us to run our own custom valuation model and aggregation algorithm, effectively allowing us to be an oracle, publishing real-world, aggregated crypto market data. There are a number of different axis of parallelism that flow naturally in this space. It is a unique challenge seeing as we are working with time-constrained scaling - we need to complete more work in a fixed amount of time. Our internal definitions of 'work' constitute actually computing and aggregating a given asset's valuation. The turnaround time available to us, is the amount of time inbetween subsequent market data events. Specifically, we parallelize the streaming of asset information, valuation computation, and aggregation of market data. With the valuation engine being the most computationally expensive part of the application, it is crucial to parallelize, especially as we scale the problem size, in order to sustain high throughput of market data while continuously running the valuation model. This is a fundamental challenge in the high frequency trading space, as it is crucial to ensure that the turnaround for the live trading flow, for each asset/exchange pair, is fast enough to sustain the exchange data throughput.

1.3 Streaming Assets Information from Exchanges

In order to get the necessary data to feed into our valuation model, we opted to use CCAPI, a header-only C++ library for streaming market data and executing trades directly from various cryptocurrency exchanges [2]. The motivation behind this is that it allowed us to quickly connect to multiple exchanges, without having to write our own connecting code. There are some important data structures and classes necessary to work with this interface. Specifically:

(1) **Event Handler**
Creating a derived instance of the Event Handler class allows us to subscribe and process Events. A single function is required:

```
virtual bool processEvent(const Event& event,
                          Session* session);
```

(2) **Event**
The Event object represents a variety of different events depending on those subscribed. We specifically handled Subscription Data events, which represent live market data event feeds.

(3) **Message**
Each event object has a message list. Each message represents a market data update from any one of the subscribed exchange/asset pairs. Each message has a map of elemnts, representing various bid/ask prices, and their respective volumes at that price. We receive a market depth of 100, indicating that we receive the top 100 orders on either side of the book.

Although the used API provides a large aspect of the functionality in terms of connecting to the various exchanges, we still maintain our own internal market data parser and internal order format, characterized by the following interface function and structures:

(1) **Market Data Parser**

```
void parse(const ccapi::Message &message,
    std::vector<Order> &bids,
    std::vector<Order> &asks);
```

(2) **Order**

```
enum OrderType { BID, ASK };

class Order {
    double price;
    double volume;
    OrderType orderType;
};
```

1.4 Valuation Model

In high frequency trading, valuation (pricing) models are fundamental to enact trading strategies. It is critical to have an internal valuation of an asset, especially when this can be skewed depending on other factors such as other market positions, hedges, exposure, and generally other relevant current events. As such, high frequency trading firms spend a lot of time and effort ensuring that their valuation engines strike a perfect balance between accuracy and speed. In industry, a multitude of different models exist, though many are proprietary. Some publicly known algorithms include midpoint average, volume weighted average price, depth weighted average price, signal valuation, etc.

1.5 Price Aggregation

Another key problem is that of aggregating these valuations across various exchanges. In both the typical equities and crypto markets, there are a large number of exchanges available for trading. While it is possible to trade independently on one exchange utilizing only the valuation computed from its market data, it is frequently necessary, and profitable to aggregate these pricing models. One simple reasoning for this is to take advantage of statistical arbitrage, where you necessarily need to concurrently have access to pricing information from more than one exchange at a time.

1.6 Input and Output

(1) **Input**
The input of our application is the streaming market data information for a variety of different exchange/asset pairs.

(2) **Output**
The output of our application is an aggregated valuation per asset, as valuated per market data update per exchange by our valuation engine.

1.7 Parallel Analysis

There are a number of different parallelization axis for the problem scope. One of the largest is that of streaming and running the valuation engine per exchange/asset pair in parallel. This is further interesting due to the locality aspects of the problem space. One such example could be running all of the computations for one asset on the same core, allowing the aggregation service to benefit from locality. Not only that, but depending on the computational cost of the valuation model, there is parallelism that can be exploited there in speeding up the computation.

In general, the key idea motivation for parallelization in the problem space is the need for high sustained throughput of market data events and their corresponding computed valuations. Late data is redundant data, so it is important that the application be able to handle a scaling of the problem size without losing that throughput, both in terms of exchange/asset pair, but also valuation engine complexity.

2 APPROACH

2.1 Technologies Used

As mentioned, in section 1.3, we utilized a CCAPI, c++ header API to aid in our streaming of asset information. From there, we implemented our own depth weighted average price valuation engine. Finally, we coalesced and aggregated results utilizing a c++ pub/sub library [3].

The parallel implementation of our application runs on 2.4 GHz 8-Core Intel Core i9 CPU on a Macbook. We target CPUs instead of GPUs because our specific application isn't well suited for GPU architecture. For one, High Frequency Trading is very latency sensitive. We are constantly running our valuation model on a stream of asset data. Since data movement is continuous, we cannot for instance, for CUDA implementation, transfer the data from host to device, run the computation, and transfer result back from device to host. If we were to queue up the data, that would add too much latency to our application. Further, GPUs are better suited for large computationally expensive calculations on multiple sets of data. Although we do have the parallel streams of data aspect, our valuation model is not computationally complex enough for it to necessitate running on a GPU. The overhead of copying the streamed information over is larger than the speedup obtained from running it on the GPU. This was definitely something we considered, however, especially with numerous lectures on architecture homogeneity.

2.2 Serial Algorithm

Our serial implementation simply subscribes to the relevant exchange/asset pair market data outlined in section 2.3, utilizing the aforementioned CCAPI [2]. Our market data event handler therefore handles events/messages by first parsing the data from CCAPI into our own internal data format, and then running our valuation engine on the data. From here, we aggregate the pricing information per asset.

2.3 Problem Space Size

One of the key aspects of our application is the actual problem space size. As with most parallel programs, the true benefits of parallelism are rarely seen at small problem sizes. This is especially true in our project. Initially, we had anticipated connecting to 5 exchanges, streaming and pricing/aggregating 5 assets from each exchange. We discovered that as we developed our application, the original number was far too small to see any benefit in terms of the sustained throughput of the parallel approach versus sequential. The combination of both a low computational complexity of our initial valuation model, as well as small number of actual exchange/asset pairs, meant that the sequential model could keep up just fine with the throughput of market data events received.

As a result, we decided to initially up the problem size, ie increasing the number of exchange/asset pairs of market data subscribed to. Our final implementation, scaled to the following exchange/asset pairs:

Exchanges - (12)
Ascendex • Binance • Binance US • Bitfinex • Bitmex • Bitstamp • Coinbase • Cryptocom • Erisx • FTX • FTX US • Kucoin
Assets - (40)
BTC • ETH • LINK • SOL • XRP • DOT • DOGE • ADA • BNB • USDC • LUNA • UST • BUSD • SHIB • WBTC • DAI • NEAR • TRX • CRO • MATIC • LTC • BLUNA • APE • FTT • LEO • ATOM • XLM • ALGO • XMR • ETC • UNI • VET • FIL • ICP • HBAR • EGLD • XTZ • THETA • SAND • GRT

This implies that our final problem size scaled from 25 exchange/asset pairs to 480 exchange/asset pairs.

2.4 Valuation Model Complexity

Another aspect of our implementation, intimately related to the problem size is the actual valuation model complexity. As outined in section 1.4, there are a number of different valuation models used in the real world, each with varying levels of computational cost. In an attempt to scale the problem size, we similarly iterated various valuation model complexities, in order to see a benefit out of parallelization. Seeing as the serial algorithm cannot proceed to the next market update until the previous exchange/asset pair valuation information is calculated, it is only natural that increasing the time spent blocking increases the latency, necessarily decreasing the overall throughput.

2.4.1 Weighted Midpoint. Our first iteration involved a simple weighted midpoint valuation model [4]. Specifically, the following:

$$\text{Valuation} = \frac{\text{bidSize} * \text{askPrice} + \text{askSize} * \text{bidPrice}}{\text{bidSize} * \text{askSize}}$$

Clearly, such a valuation model is quite basic, involving only a small number of multiplication, addition, and division operations. This is due to the fact that we only look the top of the order book, and discard any other depth. Although implemented initially, we discarded this valuation model as it was not computationally expensive enough.

2.4.2 Volume Weighted Average Price. We similarly attempted to utilize volume weighted average price as our next valuation method. VWAP is calculated by adding up the dollars traded for every transaction (price multiplied by the number of shares traded) and then dividing by the total shares traded [5].

$$\text{VWAP} = \frac{\sum Price * Volume}{\sum Volume}$$

This was an interesting approach, as we had to change the type of market data received. Rather than receiving order book updates, we instead configured our code to receive actual trade confirmations, which were much more frequent. As a result, this did become more computationally expensive on two axis. For one, we were computing more often, but more importantly, we were accumulating larger float values, causing the cost of the computation to increase. However, this similarly did not show enough of a computation to justify the need for a parallel implementation.

2.4.3 Depth Volume Weighted Average Price. We ultimately ended up implementing a depth volume weighted average price over order book updates, accounting for the actual depth of the market. Not only does this make sense from a practical perspective (top of the order book isn't necessarily the best indicator of what the market believes the price to be), but it increased our computational complexity enough to show benefits from parallelism. The key idea behind depth weighted prices is that we utilize more of the market depth, ie 2nd, 3rd, etc. depth of bids and proceed to aggregate more weighted prices. Specifically, we calculate the average price you would have to pay or receive for each quantity of the asset [6].

An example of this is the following:
E.g. Suppose the state of the market is
Bid Size | Bid | Offer | Offer Size
1 | 10 | 11 | 3
4 | 9 | 12 | 3

So if I were to buy:
1: total cost is 11

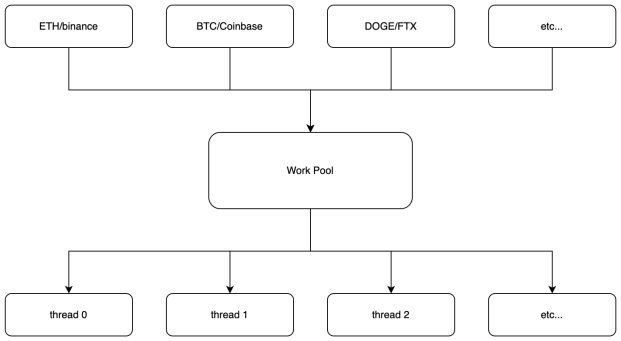

Figure 1: Dynamic Assignment

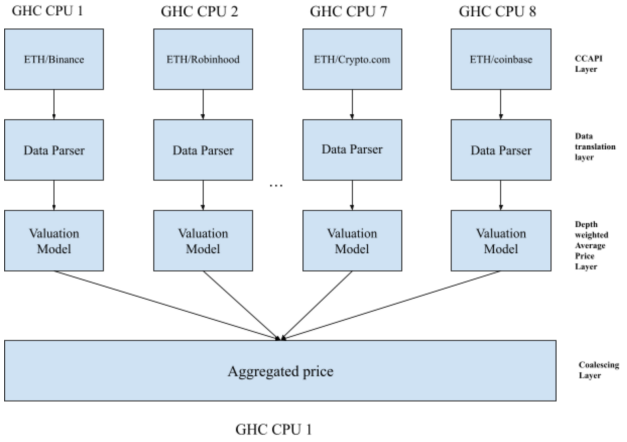

Figure 2: Static Assignment - example of application flow on one asset

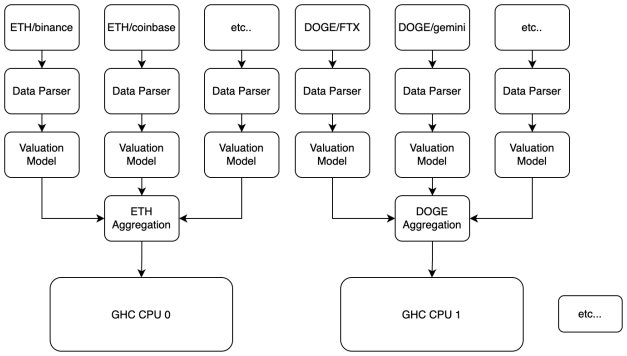

Figure 3: Static Assignment - increased problem size

2: total cost is 11
3: total cost is 11
4: total cost is 11.25
5: total cost is 11.40
Mean overall volumes: 11.13

And if I were to sell:
1: total cost is 10
2: total cost is 9.5
3: total cost is 9.333
4: total cost is 9.25
5: total cost is 9.2
Mean overall volumes: 9.457

The final price of depth volume weighted average valuation model with depth 5 is 10.2935 whereas weighted mid using only first depth is 10.5 [6]. Again, not only are we able to achieve much better accuracy using our new model, but also when utilizing higher depth and smaller volume increments, we can increase the computational complexity by a large factor.

2.5 Workload Mapping

Mapping the relevant streaming, market data parsing, valuation model, and aggregation aspects of the application to various cores was an interesting aspect of the project.

We experimented with two different work load assignment: dynamic and static.

(1) **Dynamic**
Initially, we implemented a dynamic workload assignment. We achieved this through a work pool, dynamically pulling events from the queue as they come through to the next available thread. We instantiated the work pool with 16 threads, and registered our various market data subscriptions to feed into this work pool. This meant that market data updates could arbitrarily be handled by any of the threads, depending on what is currently available at the time of the market update. Even the aggregation service fed into this work queue. A sample architecture of this workload mapping is shown in Figure 1.

(2) **Static**
We then attempted to implement a static assignment

of computational units to threads. To begin with, we implemented a version as shown in Figure 2, when we had a smaller problem size. We thought that it would make sense to evenly distributed different exchange/asset pairs to different cores, as well as the aggregated service. While this worked, we had to switch up our thought process as we scaled our problem size.

When the problem size began to scale, we began to think about locality. It would make sense that all streaming, parsing, valuation, and aggregation across exchanges for one asset occur on the same CPU, in order to take advantage of the cache locality for accessing similar data. Therefore, we arrived at a model that more closely resembles Figure 3, though at the scale of the problem size we ended up at. What this means is that each core was responsible for multiple exchange/asset pairs, though all the computation for one asset was strictly tied to one core. This allowed all valuation and aggregation aspects for an asset to stay localized to a given core.

Another static assignment we considered, that we equally theorized could give some locality benefit, was that of an exchange per core. This would have the locality benefit of ensuring that all streaming data, and the actual connection to the API stream was localized to one core. However, it had the downside of having a lot of communication across CPUs. The reason for this is that although the individual valuation models per exchange/asset pairs could be computed in parallel on each respective core, there was a larger overhead in coalescing all of the results. As such, although attempted, we ended up focusing on the aforementioned static model.

Using the standard pthread library, specifically pthread_setaffinity_np(), we were able to achieve our strict static assignment of processes to a specific CPU.

3 RESULTS

3.1 Metrics

Given the specialized domain of our application, we defined a couple of key metrics to outline the success of the project. These are outlined below:

(1) **Sustained Throughput**
Sustained throughput was one of the key, if not most important metric for the project. As outlined earlier, one of the unique challenges of the problem space is that it is time constrained. We need to perform as much work as possible in the fixed, and small amount of turnaround time we have before the next market data event arrives. Clearly, as we scale the actual problem size and introduce more exchange/asset pairs, each with varying degrees of throughput of market data events, one sequential implementation will not be able to keep up this rate. Further, as we scale the actual computational cost within our time window, the greater the need will be for parallelization.

The actual results achieved are shown below in section 3.2.

(2) **Computational Cost**
We then attempted to implement a static assignment of computational units to threads. To begin with, we implemented a version as shown

3.2 Sustained Throughput

As mentioned, sustained throughput was one of the most important metrics for our application given it's time constrained nature. One of the key first methods we utilized to improve throughput was to parallelize the streaming/parsing/aggregation of market data events per exchange/asset pairs, ultimately landing on a static assignment as shown in Figure 3. Further, we discussed the need for problem scaling, both in terms of the the number of exchange asset pairs, as well as computational cost, as determined by market depth.

We can see results of such scaling and throughput in Figure 4.

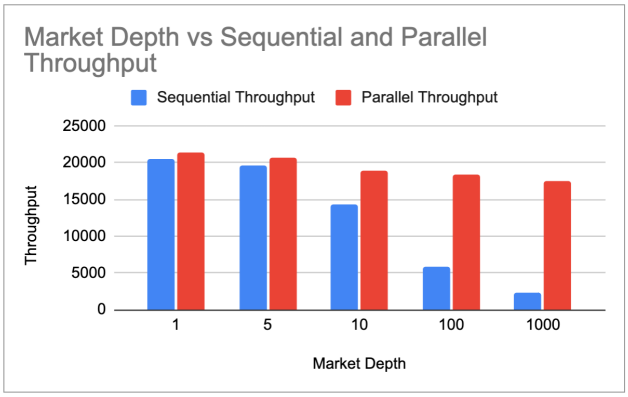

Figure 4: Sustained Throughput - Sequential vs. Parallel

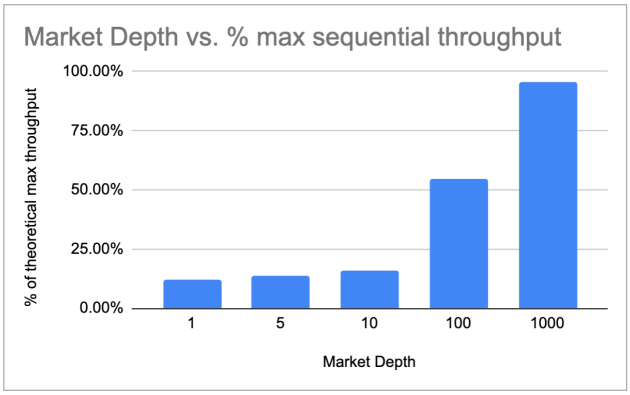

Figure 5: Percent usage of theoretical max throughput

10s, 12 exchanges, 40 assets					
Market Depth	1	5	10	100	1000
Valuation Computation Cost (ms)	58.8897	70.2466	109.6083	934.342	4233.72
Theoretical Max Sequential	169808.9819	142355.6443	91233.96677	10702.71913	2361.988984
Sequential Throughput	20571	19671	14383	5834	2252
Parallel Throughput	21362	20754	18913	18463	17463
Throughput Improvement	1.04	1.06	1.31	3.16	7.75
% of theoretical max	12.11%	13.82%	15.76%	54.51%	95.34%

Figure 6: Raw Data

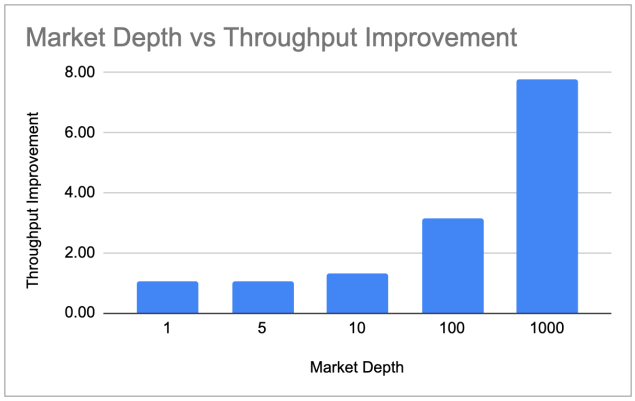

Figure 7: Throughput multiple improvement

This graph outlines the sustained throughput of our sequential implementation versus our parallel implementation. One important analysis seen from this graph is the rather meager difference in throughput between the sequential and parallel throughputs at low market depths. What this shows is that the problem size (as scaled by the market depth), in terms of the valuation computation cost was not large enough in respect to the actual throughput of market events. In other words, the sequential algorithm and its computation cost was fast enough to keep up with market data events. This is further seen in Figure 5 and Figure 6. The former graph shows the relationship between market depth and % of max theoretical throughput possible, evidently showing that the sequential version could have handled more market data events for low market depths, while reaching it's upper limit at larger ones. The latter simply shows the raw data obtained from runs of the application.

Further, figure 4 does show the effects of scaling the problem size. We can clearly see that at higher market depths, the sequential throughput clearly drops off of a cliff. Conversely, the parallel model retains the majority of its previously seen throughput. The reason for this is that as the computational cost scaled, our sequential algorithm could not perform the same amount of work in the given timespan it had between market update events. Instead, our parallel solution evidently easily keeps up with the sustained throughput rate, all while computing on more data. Another view of this data is shown in figure 7, where we can see that we only really achieve a multiplier of throughput compared to the sequential, when we begin to scale the problem size to depths 100 and 1000.

3.3 Valuation Model Computation Cost

One of the key areas of problem size scaling occurred within the valuation model computation cost. We have mentioned extensively in section 2.4 the iteration work we did in terms of upping the actual computational cost of our valuation

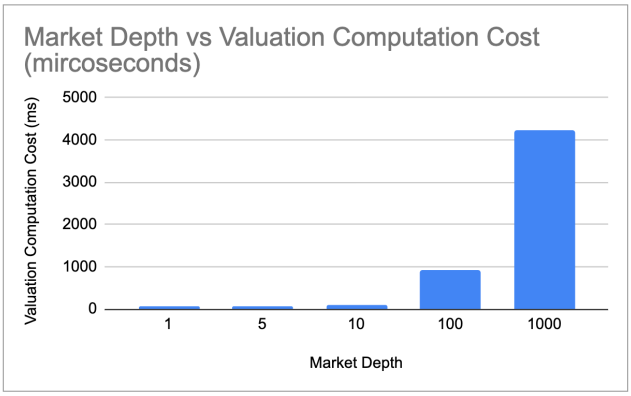

Figure 8: Computational Cost

algorithm, ultimately ending at a more accurate, let computationally intensive depth weighted average price model. This is further shown in figure 8, outlining the clear increase in computational cost at higher levels of market depth, as to be expected.

One aspect of consideration was the need to parallelize along the actual valuation computation. Although definitely something to consider, it wasn't necessarily of concern for our project scope. The reason for this is that our experimentation with our valuation model yielded a high sustained throughput, without needed to optimize within this area as well. That being said, other, more advanced valuation models, potentially incorporating machine learning strategies may be required to parallelize along this axis as well.

3.4 Profiling

Although we had a good a understanding of the bottlenecks and areas in need of parallel improvement in our code, we decided to profile it to verify our assumptions. Utilizing Instruments (as we ran the profiling on personal Mac, with a 2.4 GHz 8-Core Intel Core i9), we noticed clearly off the jump, the much higher rate of utilization of each core (as well as SMT cores) compared to our sequential model, though this was as expected (Figures 9 and 10). Further, we analyzed the percentage of our computation that was spent in the various core aspects of our application. What we found was that there was a decent overhead with the API usage. However, we explained this away due to the fact that the API took care of the actual streaming of API updates, and therefore would understandably dominate a lot of of the computation time, and therefore was a main area of parallelization focus. Further, looking at our implemented functionalities such as our own internal market data parsing and valuation models, we see the former dominating in CPU usage. This is shown in figures 11 and 10, where we have the percentages for various depths. Similarly to our previous results, we see that the valuation model computation begins to take a larger share of the percentage as we scale the problem size from market depth 100 to 1000.

WORK DISTRIBUTION

Together:

(1) Researched various cryptocurrency valuation models. Further dove into various key problems within the space, specifically within its time constrained nature.
(2) Created serial implementation of algorithm.
(3) Tested performance of final application
(4) Prepared final report and presentation.

Riccardo Santoni:

(1) Connected to various api's/exhanges to connect to actual datasets.
(2) Implemented market data parsing code.
(3) Implemented various valuation models, increasing complexity as we saw fit.
(4) Iterated and recorded data on various different valuation model complexities.
(5) Implemented static parallel assignment of threads to cores.
(6) Increased and scaled problem size.
(7) Recorded data and efficiency statstics.

Zongpeng Yu:

(1) Implemented dynamic parallel assignment of threads to cores.
(2) Recorded data and efficiency statstics.
(3) Implemented aggregation service with c++ pub sub library.

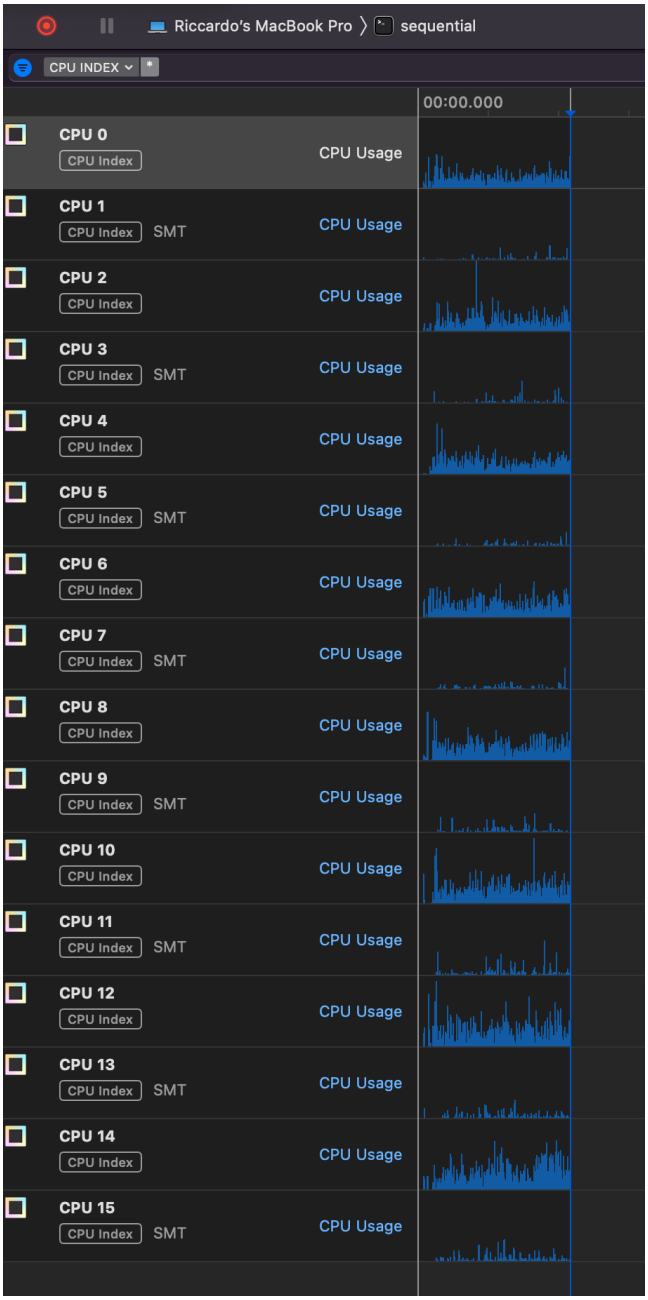

Figure 9: Sequential CPU Usage

Figure 10: Parallel CPU Usage

Figure 11: Function Percentage - Market Depth 100

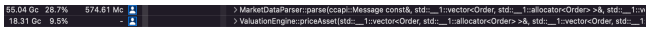

Figure 12: Function Percentage - Market Depth 1000

(4) Iterated performance of aggregation service over varying workload assignments.

Credit Distribution: 50%-50%

REFERENCES

(1) Max Dama, Q. R. (2020, May 4). Quantitative trading summary. Headlands Technologies LLC Blog. Retrieved May 5, 2022, from https://blog.headlandstech.com/2017/08/03/quantitative-trading-summary/

(2) https://github.com/crypto-chassis/ccapi

(3) https://github.com/Anil8753/CppPubSub

(4) Ganti, A. (2022, March 17). Weighted average definition. Investopedia. Retrieved May 5, 2022, from https://www.investopedia.com/terms/w/weightedaverage.asp

(5) Fernando, J. (2022, April 19). Volume-weighted average price (VWAP). Investopedia. Retrieved May 5, 2022, from https://www.investopedia.com/terms/v/vwap.asp: :text=The%20volume%2Dweighted%20average%20price%20(VWAP)%20is%20a%20trading,and%20value%20of%20a%20security.

(6) Alex CAlex C 9. (1966, October 1). Definition of mid price in literature. Quantitative Finance Stack Exchange. Retrieved May 5, 2022, from https://quant.stackexchange.com/questions/43598/definition-of-mid- price-in-literature/436084