Fast Construction of Nanosecond Level Snapshots of Financial Markets

July 22-25, 2013
XSEDE13 – San Diego, CA

Jiading Gai[1], Dong Ju Choi[2], David O'Neal[3]
Mao Ye[1] and Robert Sinkovits[2]

[1]College of Business, University of Illinois
[2]San Diego Supercomputer Center
[3]Pittsburgh Supercomputer Center

Academic debate

"High frequency trading presents a lot of interesting puzzles. The Booth [School of Business, U. Chicago] faculty lunchroom has hosted some interesting discussions: 'what possible social use is it to have price discovery in a microsecond instead of a millisecond?' 'I don't know, but there's a theorem that says if it's profitable it's socially beneficial.' 'Not if there are externalities' 'Ok, where's the externality?' At which point we all agree we don't know what the heck is going on."*

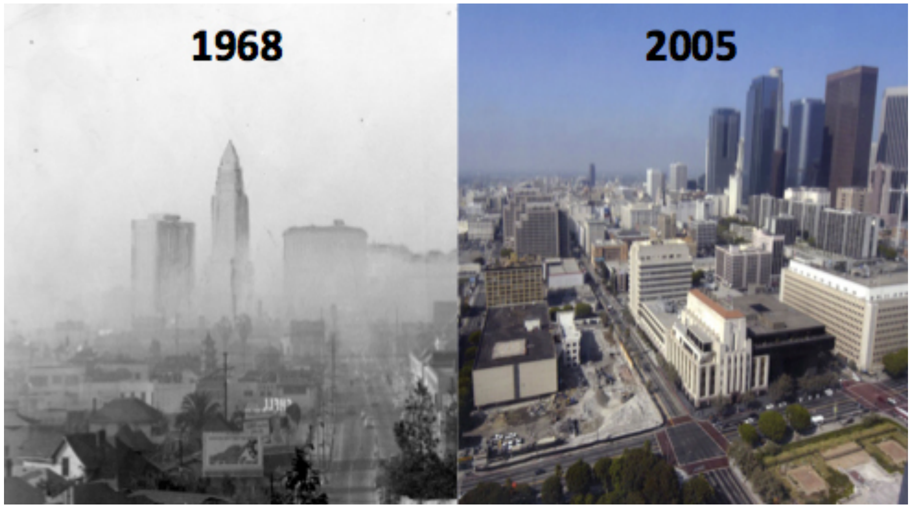
-John Cochrane

externality is a cost or benefit which results from an activity or transaction and which affects an otherwise uninvolved party who did not choose to incur that cost or benefit (Buchanan and Stubblebine, Economica **29 (116): 371–384)*

Positive and negative externalities

Amy sells all of the trees on her hillside property to Bill, who cuts down trees resulting in a mudslide that covers Carl's property

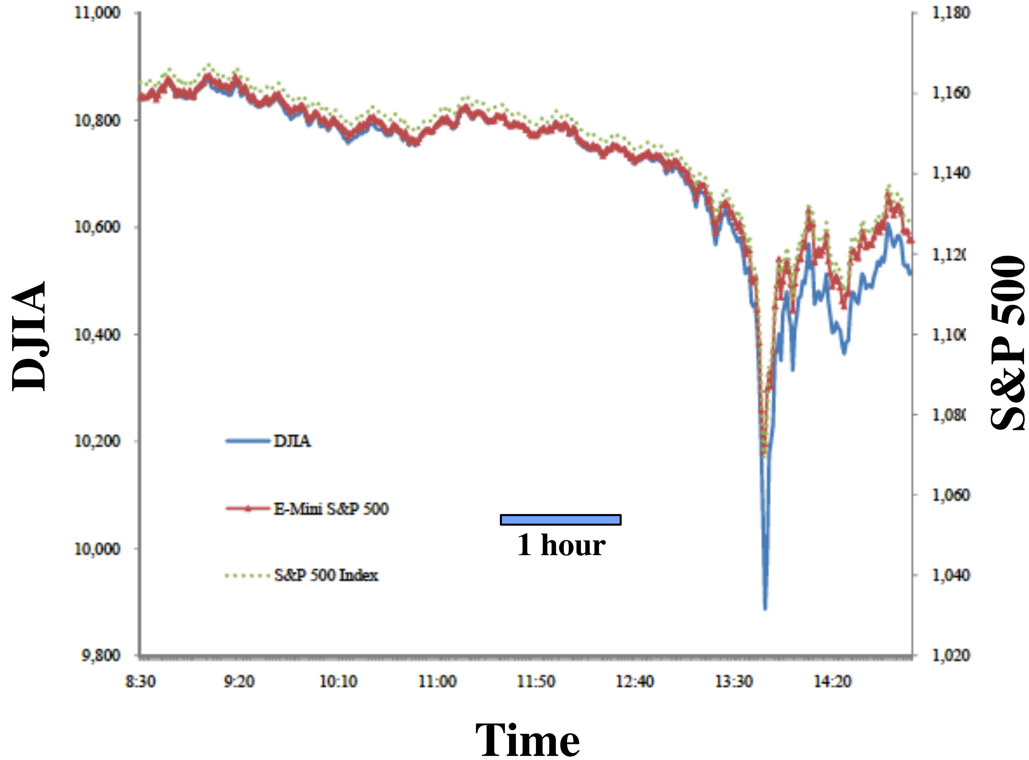
Company invests in new technology to reduce pollution, leading to better air quality for area residents



Not all high frequency trading is bad, but …

- High frequency trading at the second to millisecond timescale improves market quality (e.g. reduces spread between ask and bid prices), but decreasing latency from micro to nanoseconds does not lead to improvements of market quality measures
- Relative rank is important. When all traders have nanosecond technology, the pay-off would not be different from the case where all traders are in microseconds (Bernanke and Frank, 2012)
- Speed competition leads to higher volatility, cancellation and data size. Traders engage in unsavory behavior such as “quote stuffing” - submitting an extraordinarily large number of orders followed by immediate cancellation in order to generate order congestion (Biais and Woolley, 2011)

Negative externality example – flash crash 5/6/10



Kirilenko et al (SSRN 2011) find strong evidence implicating the activity of high frequency traders in the flash crash

“The reconstruction of even a few hours of trading during an extremely active trading day in markets as broad and complex as ours— involving thousands of products, millions of trades and hundreds of millions of data points—is an enormous undertaking. Although trading now occurs in microseconds, the framework and processes for creating, formatting, and collecting data across various types of market participants, products and trading venues is neither standardized nor fully automated.”

U.S. Commodity Futures Trading Commission and U.S. Securities and Exchange Commission: Preliminary Findings Regarding the Market Events of May 6, 2010

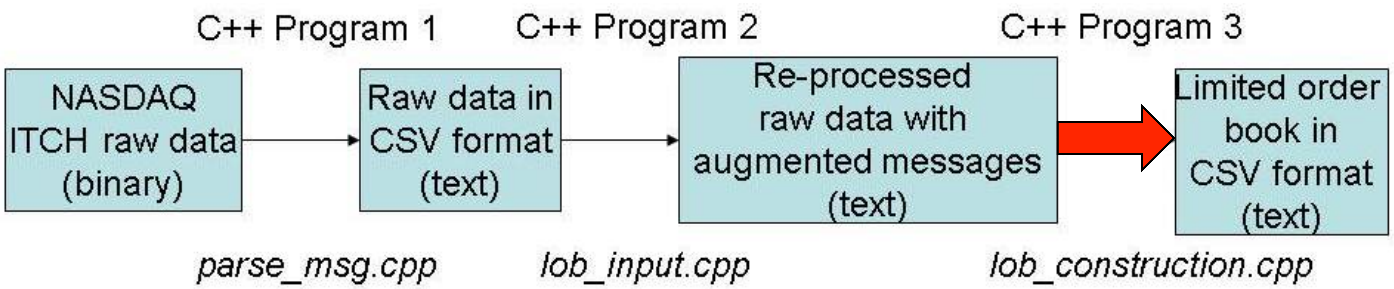
Why do we need nanosecond resolution limit order book?

To determine the impact of high-frequency trading activity on financial markets, it is necessary to construct nanosecond resolution limit order books – records of all unexecuted orders to buy/sell stock at a specified price. Analysis provides evidence of quote stuffing: a manipulative practice that involves submitting a large number of orders with immediate cancellation to generate congestion.

The traditional approach of using a <u>snapshot</u> of the limit order book is much faster, but does not have the required level of resolution to capture effects of high frequency trading. For example, snapshots taken at 9:30 and 9:35 of all unexecuted orders will miss everything that happened in between.

Data processing pipeline

Three step data processing pipeline, with run time dominated by the limit order book construction. First two steps only done once for each day of market activity and results can be used for every stock traded that day



Main data: seven types of messages

Type	Timestamp (nanoseconds)	Order Reference Number	Buy /Sell	Shares	Stock	Price	Original Order Reference Number	Market Participant ID
A	53435.759668667	335531633	S	300	EWA	19.50		
F	40607.031257842	168914198	B	100	NOK	9.38		UBSS
U	53520.367102587	336529765		300		19.45	335531633	
E	53676.740300677	336529765		76				
C	57603.003717685	625843333		100		32.25		
X	53676.638521222	336529765		100				
D	53676.740851701	336529765						

A	Add order anonymously
F	Add order with Market participant ID
U	Update: replace old order with new order
E	Order Execution
C	Order Executed with Price Message
X	Partial cancellation
D	Order Deletion

Now that we have a limit order book …

- Time Weighted Quoted spread
  - Quoted spread = (best ask-best bid)/2
  - Weighted by the time of the quote

- Size Weighted Effective Spread
  - Effective spread
    - Buy: (Transaction price-midpoint of best bid and ask)*2
    - Sell: (Midpoint of best bid and ask- Transaction price)*2
  - Weighted by transaction size

- Time weighted depth
  - Number of shares in the best bid and ask of the limit order book
  - Number of shares within 10 cents of the best bid and ask
  - Weighted by time

- 1-minute short term volatility
- 2-minute to 1-minute variance ratio

Limit order book construction can be time consuming

Symbol	Wall time (s) original code
SWN	8400
AMZN	55200
AAPL	129914

Timings obtained using all 16 cores on a single Gordon compute node (dual socket 2.6 GHz Intel Sandy Bridge). NASDAQ trading data from June 4, 2010.

Original shared memory version of code parallelized using Pthreads

Like any other task that we expect to do repeatedly, reducing time to solution will make the researchers more productive

Code optimization (part I) – do things once

Profiling code indicated that a large fraction of the run time was spent converting string to floats or integers. This was not initial I/O, but rather the repeated conversion inside inner loops. Also expending considerable time in string comparisons

```
// Operation performed inside loops
seqcurrent  = atof(settled[y][8].c_str());
seqoriginal = atof(settled[y][9].c_str());
if (settled[y][3].compare("B") == 0)
```

Before

After

```
// Do once at start of program
for (int y=0; y < numRows; y++) {
  rss[y].fset5 = atof(settled[y][5].c_str());
  rss[y].fset8 = atof(settled[y][8].c_str());
  rss[y].fset9 = atof(settled[y][9].c_str());
  rss[y].iset4 = atoi(settled[y][4].c_str());
  rss[y].isB   = settled[y][3].compare("B");
  rss[y].isS   = settled[y][3].compare("S");
}
```

```
// Then use results repeatedly
seqcurrent  = rss.fset8[y];
seqoriginal = rss.fset9[y];
if (rss.isB == 0)
```

Code optimization (part II) – avoid serialization

In main parallel loop, all threads write output to file. To avoid conflicts, locks set so that only one thread writes at a time. Unfortunately, this forces serialization. Instead, store results to array and output after exiting loop.

```
for (...) {
  rc = pthread_mutex_lock(&mutex);
  checkResults("pthread_mutex_lock()\n",rc);
  lob_msft.open(writeFile,ios::app);
  lob_msft << ...
  lob_msft.close();
  rc = pthread_mutex_unlock(&mutex);
  checkResults("pthread_mutex_unlock()\n",rc);
}
```

Critical region

Before

After

```
ostringstream accum[fRows];
for (...) {
   accum[i] << ...
}
for(...) {
  lob_msft << accum[i].str();
}
```

Parallel

Code optimization (part III) – dynamic scheduling

Main function still showing imperfect load balancing. Iterations of key loop are independent, but take different amounts of time to execute. Strip out pthreads code (dense, hard to read & maintain) and replace with OpenMP directive with dynamic scheduling

```
for (int t = 1; t<numThreads-1; t++) {
  comp[t].start = comp[t-1].end;
  comp[t].end   = comp[t-1].end+tInc;
}
for (int t = 0; t<numThreads; t++) {
  pthread_create(&threads[t],NULL,tFunc1,(void*)&comp[t]);
}
for (int t = 0; t<numThreads; t++) {
  pthread_join(threads[t],NULL);
}
```

Static

Before

After

```
// Within tFunc1
#pragma omp parallel for private(i) schedule(dynamic,10)
for (...) {
  // Expensive, but independent operations
}
```

Dynamic

Code optimization (part IV) – early exit

The iterations within the key function are not only independent, but can often be terminated early by taking advantage of data ordering.

```
for (...) { // main loop
  for(int y = numRows-1; y >= 0; y--) {
     seqcurrent  = rss[y].fset8;
     seqoriginal = rss[y].fset9;
     if ( (seqcurrent  >  macro_seqcurrent) &&
          (seqoriginal < macro_seqcurrent) ) {
        // additional code not shown
     }
  }
}
```

Before

After

```
for (...) { // main loop
  for(int y = numRows-1; y >= 0; y--) {
     seqcurrent  = rss[y].fset8;
     seqoriginal = rss[y].fset9;
     if (seqcurrent < macro_seqcurrent) break; // No need to keep going!
     if ( (seqcurrent  >  macro_seqcurrent) &&
          (seqoriginal < macro_seqcurrent) ) {
        // additional code not shown
     }
  }
}
```

Performance improvements – single securities

Symbol	Wall time (s) original code	Wall time (s) modified code	Speedup
SWN	8400	128	66x
AMZN	55200	437	126x
AAPL	129914	1145	113x

Timings obtained using all 16 cores on a single Gordon compute node (dual socket 2.6 GHz Intel Sandy Bridge). NASDAQ trading data from June 4, 2010

100x reduction in run time is a game changer

When the construction of the limit order book can take up to 36 hours for a single security, the utility of the technique is limited. We can only go back after the fact and slowly work through the trading data.

But… now that the speedups of 100x or more have been achieved, it's feasible to analyze an entire market on a daily basis.

A few things are working in our favor

- Each security is independent (2700+ way parallelism?)
- Only a small fraction of the securities are highly time consuming
- Input and other overhead can be amortized over many securities

LOB construction for the full NASDAQ

5/6/10 (2960 symbols) "Flash crash"

Step	Gordon (s)	Stampede (s)	Blacklight (s)	Memory (GB)
Parse	1315	1158	1672	33
LOB input	9705	8149	15051	62
LOB construct	31938	40495	66855	59

8/1/12 (2754 symbols) Knight Capital computer glitch

Step	Gordon (s)	Stampede (s)	Blacklight (s)	Memory (GB)
Parse	511	451	660	29
LOB input	2865	2536	4880	35
LOB construct	7885	8045	18921	44

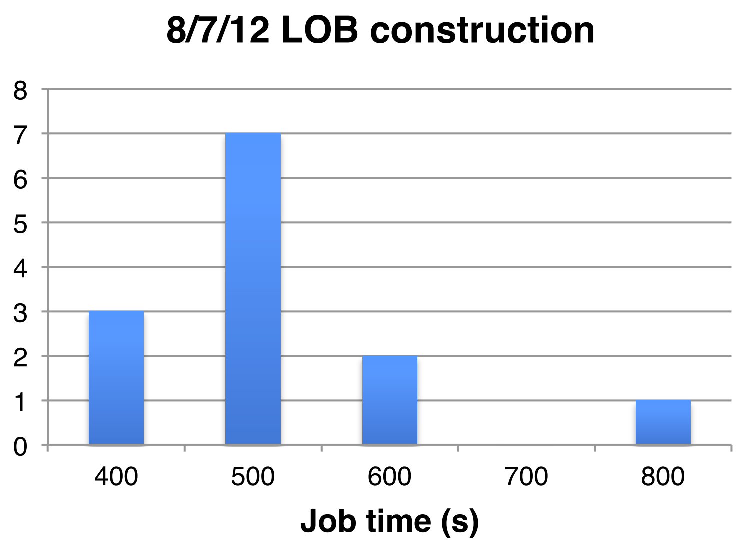
8/7/12 (2750 symbols) typical trading day

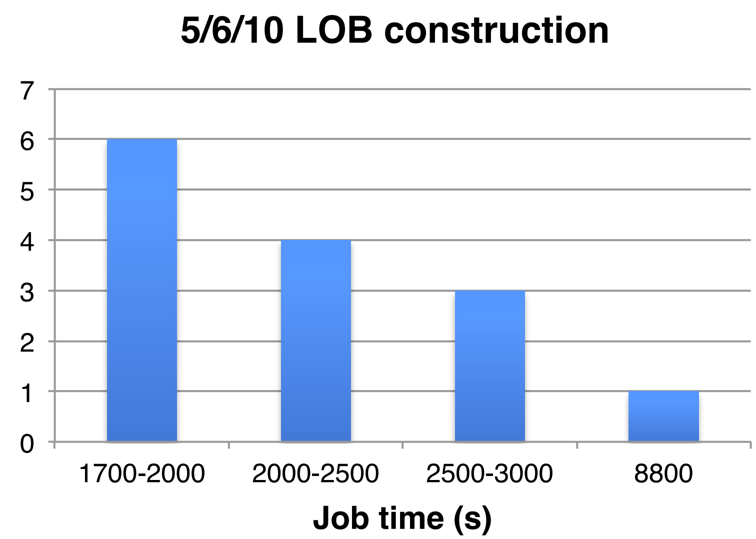
Step	Gordon (s)	Stampede (s)	Blacklight (s)	Memory (GB)
Parse	361	313	461	27
LOB input	2017	1781	3289	44
LOB construct	6005	5774	14965	29

All timings on one 16 core node or blade

A first crack at parallelism across stocks

Divide stocks into groups of 200 in alphabetical order and run as 14-15 independent jobs. Time to solution is max for all jobs





Time per stock scales roughly as square of number of records in input data. Can probably balance work reasonably well.

Outlier dominated by a single security (QQQQ), the NASDAQ-100 index, which required almost 2 hours

A new bottleneck … in a good way

For a day of typical activity, with LOB construction divided over 14-15 jobs, LOB input preparation step is the new bottleneck. LOB construction time is really limited by access to compute hardware or work needed for most time consuming stock.

8/7/12 (2750 symbols) typical trading day

Step	Gordon (s)
Parse	**361**
LOB input	**2017**
LOB construct (1 node)	**6005**
LOB construct (14 nodes)	**819**

Exchanges Agree to Add Smaller Trades to U.S. Stock Volume Count

By Sam Mamudi - Jun 24, 2013 10:27 AM PT

0 COMMENTS

QUEUE

Stock trades of fewer than 100 shares that have traditionally been excluded from U.S. share volume tallies are poised to be added to the count.

The New York Stock Exchange, the Nasdaq Stock Market and the Financial Industry Regulatory Authority Inc. agreed on a plan to add odd lots to official records of daily trading in individual stocks and the overall market, Colin Clark, NYSE Euronext senior vice president and representative to the CTA Operating Committee, said in an e-mailed message.

Cornell's O'Hara, together with Chen Yao and Mao Ye of the University of Illinois, published a paper in 2011 which estimated that excluding odd lots meant 4 percent of stock trading volume was omitted. After updating their data ahead of publication in a forthcoming issue of the Journal of Finance, the authors say that figure is 4.9 percent.