Efficient construction of limit order books for financial markets

Robert Sinkovits
San Diego Supercomputer Center
Abstract *

A limit order book (LOB) is a record of unexecuted orders to buy or sell a stock at a specified price. The LOB can then be used as a starting point for deeper analysis of markets, leading to a better understanding of the impact of trading behaviors, suggestions for regulations to make markets more effective or identification of manipulative practices such as quote stuffing. Construction of full-resolution LOBs is computationally demanding and, as a consequence, approximations are often employed. Unfortunately, this limits the utility of the LOBs in the era of high frequency trading. In this collaboration with Mao Ye (U. Illinois), we describe how we were able to first optimize the performance of existing full-resolution LOB construction software to achieve a 100x reduction in run time, and then refactor the software to ultimately improve time to solution by 1000-3000x.

* Not meant to be read now, added for completeness
History of collaboration

• Started as ECSS project back in 2012
  – Mao Ye (U. Illinois College of Business)
  – Robert Sinkovits (SDSC), David O’Neal (PSC), DJ Choi (SDSC)
• Continued under NSF EAGER funding (PI Ye, co-PI Sinkovits)
• Work presented at XSEDE 13 and XSEDE14
• In 2018, Ye and Whited co-organized National Bureau of Economics Research (NBER) workshop on “Big Data and High-Performance Computing for Financial Economics”. Invited ECSS staff to present on XSEDE and ECSS.

https://papers.nber.org/si2018_video/bigdatafinancialecon/
High-Performance Computing for Financial Economists
Anirban Jana, Pittsburgh Supercomputing Center

National Bureau of Economic Research
Summer Institute, Big Data and High-Performance Computing for Financial Economics

XSEDE Extended Collaboration Support Services
Robert Sinkovits, San Diego Supercomputer Center

ECSS areas (under the hood)
- ECSS controls of the area. As an end user, you won’t need to be aware of
  the area and your JATP will figure out where to in each
- Extended Support for Research Teams (ESRT): Traditional ECSS projects to
  support small and medium-sized projects
- Extended Support for Community Codes (ESCC): Similar to ESRT but with
  resources made available for academic use
- Extended Support for Testing Execution (ESTE): Development of workflows
  and best practices for ECSS execution
- Extended Support for Advanced Computing (ESAC): Extended support for
  research involving advanced computing
- Extended Support for Education, Training, and Research (ESTR): Technical
  support for use of advanced Computer

https://papers.nber.org/si2018_video/bigdatafinancialecon/
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.
Limit order book

- Market data consists of 17 different types of messages, describing a variety of activities. For our purposes, we’re only concerned with a few of these
  - Add new order to buy/sell with (F) or without (A) market participant ID
  - Execute order with (C) or without (E) price message
  - Delete an outstanding order, either fully (D) or partially (X)
  - Update an order (U) – replace an old order with a new one

- A limit order is an order placed on an exchange to buy or sell a certain number of shares at a price equal or better than market price. The limit order book (LOB) is just a record of unexecuted orders

- Once we have the LOB, we can calculate bid-ask spread and market depths (number of outstanding orders at each price)
### Seven types of messages that we care about

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Add order anonymously</td>
</tr>
<tr>
<td>F</td>
<td>Add order with Market participant ID</td>
</tr>
<tr>
<td>U</td>
<td>Update: replace old order with new order</td>
</tr>
<tr>
<td>E</td>
<td>Order Execution</td>
</tr>
<tr>
<td>C</td>
<td>Order Executed with Price Message</td>
</tr>
<tr>
<td>X</td>
<td>Partial cancellation</td>
</tr>
<tr>
<td>D</td>
<td>Order Deletion</td>
</tr>
</tbody>
</table>
Sample Limit Order Book
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.
LOB construction can be time consuming

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Wall time (s) original code</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWN</td>
<td>8,400</td>
</tr>
<tr>
<td>AMZN</td>
<td>55,200</td>
</tr>
<tr>
<td>AAPL</td>
<td>129,914</td>
</tr>
</tbody>
</table>

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.
Optimizing the software

In the next few slides, I’ll go through the four steps that we took to improve the performance of the software

- Do things once
- Avoid serialization
- Dynamic scheduling
- Exit loops early
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)

// 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
Before
seqcurrent = rss.fset8[y];
seqoriginal = rss.fset9[y];
if (rss.isB == 0)

After
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.

```cpp
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);
}
```

Before

<table>
<thead>
<tr>
<th>Critical region</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel</td>
</tr>
</tbody>
</table>

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

After
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);
}
```

**Before**

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

**After**

```
for (int t = 1; t<numThreads-1; t++) {
    Static
    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);
}
Code optimization (part IV) – exit loops early

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

```c
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

```c
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
        }
    }
}
```

After
## Impact of optimized code

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Wall time (s) original code</th>
<th>Wall time (s) modified code</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWN</td>
<td>8,400</td>
<td>128</td>
<td>66x</td>
</tr>
<tr>
<td>AMZN</td>
<td>55,200</td>
<td>437</td>
<td>126x</td>
</tr>
<tr>
<td>AAPL</td>
<td>129,914</td>
<td>1,145</td>
<td>113x</td>
</tr>
</tbody>
</table>
LOB construction for the full NASDAQ

### 5/6/10 (2960 symbols) “Flash crash”

<table>
<thead>
<tr>
<th>Step</th>
<th>Gordon (s)</th>
<th>Stampede (s)</th>
<th>Blacklight (s)</th>
<th>Memory (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse</td>
<td>1,315</td>
<td>1,158</td>
<td>1,672</td>
<td>33</td>
</tr>
<tr>
<td>LOB input</td>
<td>9,705</td>
<td>8,149</td>
<td>15,051</td>
<td>62</td>
</tr>
<tr>
<td>LOB construct</td>
<td>31,938</td>
<td>40,495</td>
<td>66,855</td>
<td>59</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Step</th>
<th>Gordon (s)</th>
<th>Stampede (s)</th>
<th>Blacklight (s)</th>
<th>Memory (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse</td>
<td>511</td>
<td>451</td>
<td>660</td>
<td>29</td>
</tr>
<tr>
<td>LOB input</td>
<td>2,865</td>
<td>2,536</td>
<td>4,880</td>
<td>35</td>
</tr>
<tr>
<td>LOB construct</td>
<td>7,885</td>
<td>8,045</td>
<td>18,921</td>
<td>44</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Step</th>
<th>Gordon (s)</th>
<th>Stampede (s)</th>
<th>Blacklight (s)</th>
<th>Memory (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse</td>
<td>361</td>
<td>313</td>
<td>461</td>
<td>27</td>
</tr>
<tr>
<td>LOB input</td>
<td>2,017</td>
<td>1,781</td>
<td>3,289</td>
<td>44</td>
</tr>
<tr>
<td>LOB construct</td>
<td>6,005</td>
<td>5,774</td>
<td>14,965</td>
<td>29</td>
</tr>
</tbody>
</table>
Next steps – going beyond ECSS

• Our initial ECSS project (Sinkovits, O’Neal, Choi, Ye) was quite successful, resulting in 100x reduction in time to solution

• We recognized there was more to be done, but goals were too ambitious for an ECSS project
  – Required rethinking algorithms rather than just tuning existing algorithms
  – Making optimal use of local (flash) file systems

• Work supported by NSF EAGER grant to Ye and Sinkovits
Goal of NSF funded work (SDSC component)

Extend the LOB construction to much more challenging problems (e.g. combined analysis of all stocks, across multiple markets, on very busy days of trading)

- Requires further reduction in time to solution
- New algorithms needed to reduce memory footprint so that standard, as opposed to specialized large memory, hardware could be used
- Data management strategies for working with large input data and large file counts
Strategy

• Pre-process raw market data (ITCH format) to generate separate CSV file for each stock
• Make use of local (SSD) file system to deal efficiently with large numbers of temporary files
• LOB construction is parallelized over stocks using dynamic assignment of work – time required for different stocks can vary by at least 1000x
• Serial application for LOB construction makes use of hash to keep track of unexecuted orders at each price
1. Split message data

LOB Construction workflow

Blue = parallel file system
Orange = Gordon SSD
For clarity only 16 processes are shown. Can be extended to all cores on a shared memory compute node.
1. Split message data
   - AAPL_01
   - GOOG_01
   - NVDA_01
   - AAPL_02
   - GOOG_02
   - NVDA_02
   - AAPL_15
   - GOOG_15
   - NVDA_15
   - AAPL_16
   - GOOG_16
   - NVDA_16

2. Split into stock-specific results (parallel)
For clarity only three stocks are shown. Can be extended to arbitrary number of stocks.
1. Split message data

   x01
   - AAPL_01
   - GOOG_01
   - NVDA_01

   x02
   - AAPL_02
   - GOOG_02
   - NVDA_02

   ... (continued)

   x15
   - AAPL_15
   - GOOG_15
   - NVDA_15

   x16
   - AAPL_16
   - GOOG_16
   - NVDA_16

2. Split into stock-specific results (parallel)

3. Combine partial files (parallel)

LOB Construction workflow

Blue = parallel file system
Orange = Gordon SSD
1. Split message data

- AAPL_01
- GOOG_01
- NVDA_01

- AAPL_02
- GOOG_02
- NVDA_02

- AAPL_15
- GOOG_15
- NVDA_15

- AAPL_16
- GOOG_16
- NVDA_16

2. Split into stock-specific results (parallel)

3. Combine partial files (parallel)

4. Construct LOBs (parallel)

- AAPL → AAPL_LOB
- GOOG → GOOG_LOB
- NVDA → NVDA_LOB

Blue = parallel file system
Orange = Gordon SSD
1. Split message data

Message Data

x01 → AAPL_01 → AAPL
x01 → GOOG_01 → GOOG
x01 → NVDA_01 → NVDA

x02 → AAPL_02 → AAPL
x02 → GOOG_02 → GOOG
x02 → NVDA_02 → NVDA

x15 → AAPL_15 → AAPL
x15 → GOOG_15 → GOOG
x15 → NVDA_15 → NVDA

x16 → AAPL_16 → AAPL
x16 → GOOG_16 → GOOG
x16 → NVDA_16 → NVDA

2. Split into stock-specific results (parallel)

3. Combine partial files (parallel)

4. Construct LOBs (parallel)

AAPL → AAPL_LOB
GOOG → GOOG_LOB
NVDA → NVDA_LOB

5. Archive results

TAR
Impact of second round of optimizations

New algorithms and approach to parallelization resulted in 50x speedup over previously optimized versions of the codes.

<table>
<thead>
<tr>
<th>trading date</th>
<th>tickers</th>
<th>time original (s)</th>
<th>time new (s)</th>
<th>memory original (GB)</th>
<th>memory new (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>08/07/12</td>
<td>2750</td>
<td>6005</td>
<td>320</td>
<td>29</td>
<td>0.90</td>
</tr>
<tr>
<td>08/01/12</td>
<td>2885</td>
<td>7885</td>
<td>349</td>
<td>44</td>
<td>1.16</td>
</tr>
<tr>
<td>05/06/10</td>
<td>2960</td>
<td>31983</td>
<td>663</td>
<td>59</td>
<td>1.71</td>
</tr>
</tbody>
</table>
Tackling even more challenging problems

Just when we thought that we’re done, a more challenging problem comes along – let’s analyze the combined NYSE, BATS and NASDAQ on heaviest days of trading (to date)

<table>
<thead>
<tr>
<th>Trading date</th>
<th>tickers</th>
<th>data (GB)</th>
<th>results (GB)</th>
<th>split (mm:ss)</th>
<th>LOB (mm:ss)</th>
<th>tar (mm:ss)</th>
<th>total (mm:ss)</th>
<th>memory (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>08/07/12</td>
<td>2750</td>
<td>4.9</td>
<td>14</td>
<td>2:15</td>
<td>2:20</td>
<td>0:45</td>
<td>5:20</td>
<td>0.90</td>
</tr>
<tr>
<td>08/01/12</td>
<td>2885</td>
<td>6.1</td>
<td>17</td>
<td>2:45</td>
<td>2:10</td>
<td>0:54</td>
<td>5:49</td>
<td>1.16</td>
</tr>
<tr>
<td>05/06/10</td>
<td>2960</td>
<td>14</td>
<td>37</td>
<td>7:19</td>
<td>7:25</td>
<td>2:55</td>
<td>17:39</td>
<td>1.71</td>
</tr>
<tr>
<td>05/07/10</td>
<td>7892</td>
<td>81</td>
<td>219</td>
<td>60:58</td>
<td>38:32</td>
<td>21:36</td>
<td>121:02</td>
<td>2.87</td>
</tr>
<tr>
<td>08/10/11</td>
<td>7891</td>
<td>100</td>
<td>270</td>
<td>58:58</td>
<td>36:26</td>
<td>32:41</td>
<td>119:05</td>
<td>4.02</td>
</tr>
</tbody>
</table>
Tackling even more challenging problems

Just when we thought that we’re done, a more challenging problem comes along – let’s analyze the combined NYSE, BATS and NASDAQ on heaviest days of trading (to date)

<table>
<thead>
<tr>
<th>Trading date</th>
<th>tickers</th>
<th>data (GB)</th>
<th>results (GB)</th>
<th>split (mm:ss)</th>
<th>LOB (mm:ss)</th>
<th>tar (mm:ss)</th>
<th>total (mm:ss)</th>
<th>memory (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>08/07/12</td>
<td>2750</td>
<td>4.9</td>
<td>14</td>
<td>2:15</td>
<td>2:20</td>
<td>0:45</td>
<td>5:20</td>
<td>0.90</td>
</tr>
<tr>
<td>08/01/12</td>
<td>2885</td>
<td>6.1</td>
<td>17</td>
<td>2:45</td>
<td>2:10</td>
<td>0:54</td>
<td>5:49</td>
<td>1.16</td>
</tr>
<tr>
<td>05/06/10</td>
<td>2960</td>
<td>14</td>
<td>37</td>
<td>7:19</td>
<td>7:25</td>
<td>2:55</td>
<td>17:39</td>
<td>1.71</td>
</tr>
<tr>
<td>05/07/10</td>
<td>7892</td>
<td>81</td>
<td>219</td>
<td>60:58</td>
<td>38:32</td>
<td>21:36</td>
<td>121:02</td>
<td>2.87</td>
</tr>
<tr>
<td>08/10/11</td>
<td>7891</td>
<td>100</td>
<td>270</td>
<td>58:58</td>
<td>36:26</td>
<td>32:41</td>
<td>119:05</td>
<td>4.02</td>
</tr>
</tbody>
</table>

This would have taken 2400-7200 hours using the original version of the code on a large memory (> 64 GB node)
Summary

• ECSS and NSF EAGER funded portions of project were extremely successful
• PI can now handle problems that were previously considered to be intractable
• Economics and finance community is excited to work with XSEDE, using both HPC resources and expertise