Performance Engineering of a Trading Exchange’s Risk Management System

Rajesh Mansharamani, Manoj Nambiar
rajesh.mansharamani@swperfengg.com, m.nambiar@tcs.com

Conventional IT system architectures revolve around Web/App/Database tiers. This is not the right choice for compute intensive requirements which lend themselves to High Performance Computing (HPC) architectures. While the engineering and scientific communities are mature on the HPC front, this expertise is sorely lacking in IT system managers. This is where software performance engineering has a major role to play. We illustrate the point through a case study where common sense performance principles were applied to speed up a risk management system by several orders of magnitude.

1. Introduction

Exchanges need to perform risk computations to ensure that trading members and clearing members have sufficient collateral for clearing and settlement of trades. A number of risk computations are done to compute a variety of margins for future contracts [WIKI2010a, WIKI2010b, and CMEG2010].

We were initially faced with the task of capacity planning for a risk management system for a commodities exchange. The system was to be implemented as a traditional IT system with an application and a database tier and the question posed by the customer was whether a two CPU application server and a two CPU database server would suffice. Given just one day for arriving at the capacity in the absence of business domain knowledge was next to impossible. It took us three weeks to understand the risk management computations involved and a simple spreadsheet model estimated that 47 billion computations per second were required to be done!

This clearly was too much for a database server to handle and too much for a loosely coupled architecture spanning across application and database servers. Instead in-memory computations were to be done. A small prototype was written in C and tests were done across CPUs from leading vendors. Even with a full in-memory compute solution and assuming simple computations of the prototype, 1000 CPUs were required, since on the average the state of the art CPUs could process around 50 million computations per second.

Clearly the customer wasn’t impressed to see expectations raised from 2 x 2CPUs to 1000 CPUs. Neither was the customer impressed to see that this really called for an HPC (High Performance Computation) architecture which is common in the scientific and engineering space through the use of supercomputers and compute grids. The customer wanted something closer to a traditional IT system with a few commodity servers connected over a standard Ethernet switch. Massive number of servers over a proprietary and fast interconnect wasn’t called for at all.

After all attempts to convince the customer at an HPC investment failed we had two choices. Either we had to give up or take a crack at the in-memory prototype and provide simple enough performance engineering optimizations that could be digested by the designers and developers hoping that we could magically scale to several orders of magnitude. We took up the latter challenge towards helping the exchange scale to their requirements on a couple of commodity servers. This paper shares our experiences in meeting this challenge. Surely, traditional scientific computing experts in the HPC space would laugh at this proposition, but we are of
the opinion that a lot can be solved by simple common sense.

An overall treatment of the subject matter of risk management is outside the scope of this paper and we do not wish to make matters difficult for the reader. So instead we propose a simple problem statement in Section 2 that forms a subset of the computations and brings out the common set of compute challenges that we faced across all the risk management computations. Section 3 provides details of the compute environment and benchmarking strategy. Section 4 provides details of the performance engineering optimizations that resulted in more than a 1000 fold jump from where we started. Section 5 summarizes the results for the overall problem which had more complex computations as opposed to the one described in Section 2 and also provides concluding remarks.

2. Profit & Loss (P&L) Computation

In this section we introduce a simple to understand Profit and Loss Computation that not only forms a subset of the overall risk management computations that we required but also brings out the common set of challenges faced in all of our risk management computations.

Consider a trader A, who has bought and sold stocks in a stock exchange as per Table 1.

<table>
<thead>
<tr>
<th>Time</th>
<th>Trader</th>
<th>Transaction</th>
<th>Stock</th>
<th>Quantity</th>
<th>Price</th>
<th>Total Amount</th>
<th>Counter-Party</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>A</td>
<td>BUY</td>
<td>S1</td>
<td>100</td>
<td>30</td>
<td>3000</td>
<td>B</td>
</tr>
<tr>
<td>t2</td>
<td>A</td>
<td>BUY</td>
<td>S2</td>
<td>200</td>
<td>25</td>
<td>5000</td>
<td>C</td>
</tr>
<tr>
<td>t3</td>
<td>A</td>
<td>SELL</td>
<td>S1</td>
<td>40</td>
<td>35</td>
<td>1400</td>
<td>B</td>
</tr>
</tbody>
</table>

The last column refers to the counter party of the trader. For example, Trader A bought 100 shares of S1 at price 30. These were sold by Trader B who is the counter party of Trader A in the first trade in Table 1. Whenever A buys stocks he needs to make a payment and whenever he sells he will receive payment. At time t1 A has bought stocks worth 3000 and they are valued at 3000. Therefore his profit is zero. If we look at time t3, however, the price of stock S1 has increased to 35. Therefore the profit for A with respect to stock S1 at time t3 is the sum of transaction values plus the value of the net positions of the stock he is holding. This equates to: 

\[-3000 + 1400 + 60 \times 35 = 500.\]

In this way the profit of A can be computed across all stocks at a given time t as:

\[\text{Profit}(A, t) = \sum (\text{Transaction Values up to time } t) + \sum (\text{Netpositions}(A, s, t) \times \text{LTP}(s, t)) \] (1)

where the transaction value is the buy (quantity times price) or sell value at the time of a trade, with buy being negative and sell being positive, Netposition(A, s, t) is the net number of stocks of stock s that Trader A is holding at time t (this can be negative in some markets), and LTP(s, t) is the last traded price of stock s at time t in the market.

Given a sequence of trades it is quite easy to compute profit for each trader using formula (1). The biggest challenge is in keeping the profit updated with every price change. For example, consider a situation where an actively traded stock has holdings across 100,000 traders. Then for every trade the profit of all these traders has to be updated. If the stock trades hundred times per second, we need to update the profit 10 million times a second.

A simple pseudo code for updating the profit computation at a trader level is shown in Table 2. The outer loop essentially updates the total transaction values and total positions held of the buyer and the seller for the given stock which has been traded. In the inner loop we essentially apply equation (1). For doing it a bit smartly we first subtract the profit of a trader on a given stock before the trade takes place. Then we add the profit of the trader after the trade takes place.

The pseudo code in Table 2 was the baseline for us. It was implemented in C. When compiled with a gcc compiler on an Intel (Xeon 2.8Ghz) Linux server and executed as a single thread with the trade file loaded in memory the computation clocked an average of 190 trades/sec for a file of 1 million trades. The computation had a maximum of 100000 traders across 100 stocks.

<table>
<thead>
<tr>
<th>Table 2: Pseudo Code for Baseline Profit Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>int profit[MAXTRADERS]; // array of trader profits</td>
</tr>
<tr>
<td>int netpos[MAXTRADERS][MAXSTOCKS]; // net positions per stock</td>
</tr>
<tr>
<td>int netval[MAXTRADERS][MAXSTOCKS];</td>
</tr>
</tbody>
</table>
We had built a model that estimated the real risk management computation problem to be around 20 times as complex as the P&L problem described in this section. Also in reality the problem size was for 1 million traders. The client wanted a rate of 300 trades/sec to be supported. All this would mean at least a 20 x 10 x 300/190 = 315 fold performance improvement requirement in our baseline P&L computation problem given the initial implementation of Table 2.

3. Test Environment and Test Strategy

All testing was done on an Intel Nehalem EP server that had 8 Xeon 5560 cores of 2.8Ghz each and 8GB RAM. This machine was not the one intended for use at the client site but was the one available in our performance lab for test purposes. The CPU configuration was at par with other Intel servers available for the real life deployment. The operating system was CentOS 5.3 (Linux kernel 2.6.18). All programming was in C. We used both gcc and the Intel C compiler for compiling the programs. The usual mode of execution was as a single threaded program instance, except for the last tests which used OpenMP [OARB2010]. OpenMP provides simple directives to parallelize one’s code. More details will be provided in Section 4.

The input to our profit computation program was a trade file comprising of 1 million trades. This was synthetically generated through a trade generation program where one could specify the number of trades, the percentage of buy and sell orders, the distribution of stocks (including “hotset” distributions for hot stocks), the number of traders, and the price distribution per stock.

For the purpose of testing our profit computation program we used the following parameters for trade generation:
- Number of trades: 1 million
- Number of traders: 100,000
- Number of stocks: 100
Distribution of trades across stocks:
- 20% of trades on stocks 1 to 30
- 20% of trades on stocks 31 to 90
- 60% of trades on stocks 91 to 100
The distribution was used based on our experiences with a real life exchange where more than 50% trades occurred on just 5 to 10 stocks. The exchange too had a categorization across three ranges of hot, medium and cold stocks.

The trade file was loaded in memory and occupied just 20MB of RAM before any test could start.

4. Profit Computation Optimizations

4.1 Baseline Test

As discussed above the initial implementation was as per Table 2 and compiled using gcc. A

1 The inner loop in Table 2 works out to 4 computations plus loop counter increment plus 3 assignments, which we equate to 8 computations per cycle. For 1 million traders in reality, roughly 20 times complexity of application with respect to simple P&L computation in Table 2, and 300 trades per second our overall requirements work out to roughly $8 \times 1 \text{ million} \times 20 \times 300/\text{sec} = 48 \text{ billion computations/sec}$. The 20 fold complexity is a rounded off figure. In reality we needed 47 billion computations/sec.
single threaded execution clocked 190 trades/sec.

4.2 Use of Compiler Optimizations

We experimented with the optimization flags in gcc and with the –O3 optimization level gained a 70% improvement to clock 323 trades/sec. This was no surprise since the program was compute intensive and compilers are expected to a better job of optimizing such programs as opposed to I/O intensive programs.

4.3 Transpose order of traders and stocks

If we notice Table 2 for each trade the inner loop gets executed, where the inner loop runs across all traders. Since we are dealing with profit computations at the trader level it was natural for developers to declare all arrays with their first dimension being trader.

Now observe the two dimensional arrays:

```c
int netpos[MAXTRADERS][MAXSTOCKS]; // net positions per stock
int netval[MAXTRADERS][MAXSTOCKS]; // net transaction values
int profitperstock[MAXTRADERS][MAXSTOCKS];
```

With the inner loop running across traders this means that all the first dimensions will be scanned. However, C stores two dimensional arrays essentially as a single dimensional array with the first row occupying the first set of elements, the second row the second set of elements and so on and so forth. With this array storage structure, say if MAXSTOCKS is 100, we are accessing elements 1, 101, 201, ... as far as the memory is concerned. In essence we do not exploit any benefits of caching.

Given that most multicore architectures these days have L1, L2 and L3 cache it would be appropriate to exploit the benefits of caching by having array elements being accessed in the inner loop be next to each other.

Since all trader profits are being updated for a given stock it makes best sense to transpose the matrix so that for a given stock in first dimension we run through all traders in the second dimension of the array. The nice part is that there is no change to the code other than the following declarations:

```c
int netpos[MAXSTOCKS][MAXTRADERS]; // net positions per stock
int netval[MAXSTOCKS][MAXTRADERS]; // net transaction values
int profitperstock[MAXSTOCKS][MAXTRADERS];
```

This change resulted in improved caching and boosted the performance from 323 trades/sec to 4750 trades/sec, a 14.71 times jump from where we stood last and a 25 times jump from where we started originally!!

4.4 Use of Different Compilers

Given that we used an Intel server it made sense to try out the Intel C compiler [INTE2010] in addition to gcc. In reality this made better sense as Step 4.3. In real life it is never the case that what you need is readily available, and while the Intel C compiler was being procured and setup, we did the optimization in Step 4.3.

With the Intel C compiler we tried the same – O3 optimization level (icc –O3) which resulted in a 5% performance boost. This did not seem worth the investment. So we got in touch with Intel and tried out other options until we narrowed down upon the fast compilation flag (icc –fast). This resulted in a 37% improvement over Step 4.3 and without any further change of code we were clocking 6850 trades/sec. All results in the sections to follow are with the –fast option with the Intel C compiler.

4.5 Use of Partial Sums

If we look at equation (1) we are essentially decomposing the profit of a trader into two parts. First the sum of all (net) transaction values (across all stocks) and then the sum of all (net) positions in a stock times the latest stock price. In our implementation up to now we have maintained the net transaction values per trader per stock, and then added it back to the total profit for a trader. Since we are only interested in the total profit, it makes better sense to maintain the sum of net transaction values across all stocks for a given trader. This will reduce the space required from a two dimensional array of 100,000 x 100 to a single array of 100,000 elements thus improving caching as well. We can do the same also by keeping a sum of net positions per stock times the latest stock price.

We therefore introduce two arrays:

```c
int sumnetval[MAXTRADERS]; // sum of net transaction values
int sumposval[MAXTRADERS]; // sum of positions x stock prices
```
and we also maintain the latest stock price per stock as:

```c
int ltp[MAXSTOCKS];
```

### Table 3: Use of Partial Sums

```c
int profit[MAXTRADERS];
// array of trader profits
int netpos[MAXSTOCKS][MAXTRADERS];
// net positions per stock
int sumnetval[MAXTRADERS];
// sum of net transaction values
int sumposval[MAXTRADERS];
// sum of net positions x stock prices
```

```c
loop forever
1. t = get_next_trade();
2. tradevalue = t.quantity * t.price;
3. Subtract tradevalue from sumnetval[t.buyer];
4. Add tradevalue to sumnetval[t.seller];
5. Add t.quantity to netpos[t.buyer][t.stock];
6. Subtract t.quantity from netpos[t.seller][t.stock];
7. Add t.quantity * ltp[t.stock] to sumposval[t.buyer];
8. Subtract t.quantity * ltp[t.stock] from sumposval[t.seller];
9. loop for all traders r
   a. sumposval[r] = sumposval[r]+ netpos[t.stock][r] * (t.price – ltp[t.stock]);
   b. profit[r] = sumnetval[r]+ sumposval[r];
end loop
10. ltp[t.stock] = t.price;
end loop
```

We had mentioned that all stocks will not be hot. In the test input 60% of trades occur on hot stocks that have a lot of traders. The cold stocks are unlikely to have a large number of traders holding them. Representing this as a sparse matrix adds to code complexity. A lazier option at the expense of more memory is to simply skip the computation in the inner loop of Table 3, if the net position of any trader on any stock is zero. The inner loop now becomes:

```c
loop for all traders r
  if (netpos[t.stock][r] != 0) then
    a. sumposval[r] = sumposval[r]+ netpos[t.stock][r] * (t.price – ltp[t.stock]);
    b. profit[r] = sumnetval[r]+ sumposval[r];
  endif
end loop
```

Adding this extra check results in reduced computations and this provided a further increase of 12% in performance. We were now at 10,800 trades/sec and for the first time had crossed the 4 digit throughput barrier.

### 4.7 Sparse Matrix Representation

Given the improvement in Step 4.6 we were certain that we could achieve better performance by using a sparse matrix so that the number of traders scanned for cold stocks would be much smaller than going through all the traders and checking if they had positions in the stock or not.

We did not wish to complicate matters by using linked lists with pointers. Usually sparse matrices are used to conserve memory. In our case memory was not a problem with the regular two dimensional arrays. But we wanted the benefits of shortened scan cycles. So we, in fact, used two 2 dimensional arrays to achieve this purpose. The first one just maintained an activity flag. Whenever a trader had any trade on a given stock we marked the trader’s entry for that stock as 1 in the flag array. Then we maintained a list of traders per stock in the second two dimensional array. This was an unordered list. Whenever a new trader was detected with activity (that is the flag array had a zero value before the trade) the flag array was updated and then the trader’s id was added as the next item in the flag array.
list of traders. This needs to be done only in the outer loop.

Figure 1 provides an example of the flag table and the sparse matrix representation for a set of trades.

<table>
<thead>
<tr>
<th>Trade No.</th>
<th>Buyer</th>
<th>Seller</th>
<th>Stock</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>B</td>
<td>s1</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>C</td>
<td>s2</td>
</tr>
<tr>
<td>3</td>
<td>D</td>
<td>E</td>
<td>s3</td>
</tr>
<tr>
<td>4</td>
<td>E</td>
<td>A</td>
<td>s3</td>
</tr>
<tr>
<td>5</td>
<td>B</td>
<td>C</td>
<td>s2</td>
</tr>
</tbody>
</table>

Flag Table (updated/accessed in outer loop)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>s3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

List of Traders Per Stock (updated in outer loop, accessed in inner loop)

<table>
<thead>
<tr>
<th>s1</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>s2</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>s3</td>
<td>D</td>
<td>E</td>
</tr>
</tbody>
</table>

Figure 1: Sparse Matrix Representation

The inner loop of our profit computation simply went through this list of traders per stock, and if the net position was not zero it computed the profit. Note that the order of traders does not matter in the loop, so this list approach is fine. Also, one may wonder why we repeated the non-zero net position check. This is because we only stored the list of traders that had activity on any stock. It could happen that a trader bought 500 shares in a stock and sold them off, the flag array would still be 1 but the net positions would be zero.

As expected the sparse matrix optimization provided a tremendous boost of 3.24 times that of Step 4.6. We now achieved 35,000 trades/sec!

For the cold stocks this approach is fine, but for the hot stocks one may wonder if it is an overkill. We therefore kept a threshold parameter. If the number of traders in the list exceeded the threshold we would revert to the original (dense matrix) way of computation as in Step 4.6, and if the number of traders for a given stock fell below threshold we would use the sparse matrix approach. We got a slight boost with this approach and achieved 36,000 trades/sec.

4.8 Clustering of Arrays

By now we have seen that it pays to keep optimizing the inner loop that runs across traders for a given stock. We notice that apart from the netpositions array, the inner loop (see Table 3) comprises of access and updates to three single dimensional arrays:

```c
int profit[MAXTRADERS];
// array of trader profits
int sumnetval[MAXTRADERS];
// sum of net transaction values
int sumposval[MAXTRADERS];
// sum of net positions x stock prices
```

For a given trader we access the corresponding elements of these arrays. However, there is no guarantee that any cache pre-fetching by the hardware on subsequent trader elements is going to help since the sparse matrix implementation can have a different sequence of traders being accessed than the order of storage. We could gain, however, if any access to a trader element in any of these three arrays pre-fetched data from the other two arrays for the trader. This would not be possible in the way the arrays are declared.

So instead, we went for array clustering by declaring these three quantities as part of a single structure and then declaring an array of structures as:

```c
struct traderrec {
    int profit;
    int sumnetval;
    int sumposval;
}
```

struct traderrec traderinfo[MAXTRADERS];

In Step 4.7 we had specified a separate two dimensional array to store the list of traders per stock (sparse matrix implementation). We repeated the optimization of clustering by clustering the net positions of traders per stock with this list. Hence in the same structure access we could not only get the next trader to be processed but also the net positions for the trader.

This clustering turned out to be very beneficial providing us with a further boost of 94% from
Step 4.7. We now achieved 70,000 trades/sec, more than 350 times from where we started (Step 4.1).

4.9 Precompute Price Difference

If we look at the inner loop in Table 3 we notice that we have:

\[ \text{sumposval}[r] = \text{sumposval}[r] + \text{netpos}[t.\text{stock}][r] \times (t.\text{price} - \text{ltp}[t.\text{stock}]) \]

It is quite obvious that the term in brackets \((t.\text{price} - \text{ltp}[t.\text{stock}])\) has nothing to do with the inner loop that runs across traders. We moved this to outside the loop and replaced it by a variable that stored this precomputed difference. This optimization yielded a 7% gain bringing our throughput to 75,000 trades/sec.

In hindsight this optimization should have been done much upfront but for some reason we missed it. (And the compiler optimization also missed it!)

4.10 Loop Unrolling

By this time we had virtually tried out all optimizations that we could think of and did not wish to have any major code changes. We reviewed with Intel and they suggested using loop unrolling [WIKI2010c] for the inner loop. Loop unrolling essentially tries to reduce the overhead of the loop control instructions by repeating iterations of the loop, which increases the code size. All we had to do was to add one statement just before the inner loop:

```c
#pragma unroll
```

This optimization gave us 7% increase in performance bringing the computation speed to 80,000 trades/sec.

4.11 Batching of Trades

Thus far we only focused on technical optimizations. After all the purpose of these computations is for the risk management system to identify defaulters who do not have enough collateral with the exchange. This links to an alert management system. The alert management system is manually monitored and if an alert is generated the control user looks in to it and then decides to suspend a trader from further trading. The entire process takes a few seconds and the trader is allowed to continue to trade in these few seconds. (The risk management system was a post trade one.) We posed the question to the client whether they were ready to live with a small lag (of the order of a second) between the risk management system and the alert management system. We told them that such a small lag could save them a lot of capacity.

The client had their external consultant review this requirement and he was agreeable to the change since anyway traders could trade for several seconds even after alerts had been generated. What this meant for us was to look at the trades in batches of size \(n\), where \(n\) was the lag tolerable by the client. While each and every trade went in to position updates, the price changes for a stock which is what the inner loop is about, were used only at the end of every batch. Note that a batch of size \(n=1\) means zero lag at the cost of slower performance, which is what was being done until this step.

More specifically, for a batch of \(n\) trades we update the net positions in the first parse and during the parse we keep updating the \(\text{ltp}\) (last traded price) of the stock being traded. At the end of the batch we have all the net positions updated but now we use only the last \(\text{ltp}\) in the batch for any given stock. We use this last \(\text{ltp}\) in the inner loop.

Again this means moving more to the outer loop and reducing the frequency of calling the inner loop. For a batch size of 100 this resulted in a throughput of 150,000 trades/sec (88% improvement from Step 4.10) and for a batch size of 1000 we clocked 400,000 trades/sec (further boost of 267%).

4.12 Use of Parallel Processing (OpenMP)

Up till this point we did not focus on parallel processing. We wished to extract as much as possible from optimizing sequential code for performance because that is much easier for optimizing and debugging. We also did not want a complex parallel processing scheme that would make it difficult for developers and testers to maintain and test what we had implemented.
Since both gcc and Intel C compiler had support for OpenMP we decided to use it. This meant keeping just one #pragma in the code before the region to be executed in parallel. To make it easy for the compiler to optimize the code efficiently for parallelism we thought it best to parallelize across traders in chunks. In reality the system would anyway need to scale with increase in traders and the most naturally way to scale horizontally by adding servers would be to have a range of traders per server. So we thought that why not do the same within the server too. Thus we broke up the traders in to ranges of size NUMRANGE each. This was done by taking the two dimensional array used to get a list of traders to be scanned in the sparse matrix implementation and adding a third dimension in the front as NUMRANGES. So now the inner loop was a nested loop of two loops – first going through the number of ranges and then the regular inner loop to scan all traders in the given range.

With this optimization and using the OpenMP directive that allows for chunks of work to be scheduled (where one chunk is one range of traders) we achieved a throughput of 1,020,000 trades/sec for a NUMRANGE = 64 (with around 16000 traders per range). We set the threading to 32 threads on the 8 core Intel Nehalem EP server.

### 4.13 Summary of Optimizations

Table 4 summarizes the optimizations and their benefits. This shows that a lot (5000 fold improvement) can be achieved through simple performance engineering.

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Thousand Trades/sec</th>
<th>Improvement</th>
<th>Overall Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Baseline</td>
<td>0.19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 gcc -O3</td>
<td>0.32</td>
<td>1.70X</td>
<td>1.7X</td>
</tr>
<tr>
<td>3 Transpose of traders and stocks in two</td>
<td>4.75</td>
<td>14.70X</td>
<td>25X</td>
</tr>
<tr>
<td>4 Intel C compiler – fast optimization</td>
<td>6.85</td>
<td>1.37X</td>
<td>36X</td>
</tr>
<tr>
<td>5 Use of Partial Sums</td>
<td>9.65</td>
<td>1.41X</td>
<td>50X</td>
</tr>
<tr>
<td>6 Skip Zero Values</td>
<td>10.80</td>
<td>1.12X</td>
<td>56X</td>
</tr>
<tr>
<td>7 Sparse Matrix Optimization</td>
<td>35</td>
<td>3.24X</td>
<td>184X</td>
</tr>
<tr>
<td>8 Threshold dense versus sparse usage</td>
<td>36</td>
<td>1.03X</td>
<td>189X</td>
</tr>
<tr>
<td>9 Clustering of Arrays</td>
<td>70</td>
<td>1.94X</td>
<td>368X</td>
</tr>
<tr>
<td>10 Precompute Price Difference</td>
<td>75</td>
<td>1.07X</td>
<td>394X</td>
</tr>
<tr>
<td>11 Loop Unrolling</td>
<td>80</td>
<td>1.07X</td>
<td>421X</td>
</tr>
<tr>
<td>12 Trade Batch Size 100</td>
<td>150</td>
<td>1.88X</td>
<td>789X</td>
</tr>
<tr>
<td>13 Trade Batch Size 1000</td>
<td>400</td>
<td>2.67X</td>
<td>2105X</td>
</tr>
<tr>
<td>14 OpenMP</td>
<td>1020</td>
<td>2.55X</td>
<td>5368X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>dimensional arrays</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>4 Intel C compiler – fast optimization</td>
<td>6.85</td>
</tr>
<tr>
<td>5 Use of Partial Sums</td>
<td>9.65</td>
</tr>
<tr>
<td>6 Skip Zero Values</td>
<td>10.80</td>
</tr>
<tr>
<td>7 Sparse Matrix Optimization</td>
<td>35</td>
</tr>
<tr>
<td>8 Threshold dense versus sparse usage</td>
<td>36</td>
</tr>
<tr>
<td>9 Clustering of Arrays</td>
<td>70</td>
</tr>
<tr>
<td>10 Precompute Price Difference</td>
<td>75</td>
</tr>
<tr>
<td>11 Loop Unrolling</td>
<td>80</td>
</tr>
<tr>
<td>12 Trade Batch Size 100</td>
<td>150</td>
</tr>
<tr>
<td>13 Trade Batch Size 1000</td>
<td>400</td>
</tr>
<tr>
<td>14 OpenMP</td>
<td>1020</td>
</tr>
</tbody>
</table>

### 5. Relevance to Risk Management and Conclusions

As explained earlier, all the optimization results naturally apply to the risk management computations that were part of the original problem (too complex in functionality to explain here). But using the same optimization techniques as outlined in Section 4 we were able to scale each risk management module to 300,000 trades/sec (single threaded) for the same trade file and same parameters being tested (100,000 traders and 100 stocks). We had six such modules which meant a total single threaded performance of 50,000 trades/sec per range of 100,000 traders. We could either split the total number of traders across multiple low end boxes or use OpenMP within a box or do nothing since our target for the client was just 300 trades/sec for 1 million traders.

As it turned out the client project was accepted thanks to the performance engineering. The overall architecture involved 4 x 2 CPU nodes to account for communication overheads and fault tolerance, as well as to provide headroom for the future. Each node had a mirror node that did the same set of computations and took
over immediately upon failure of the primary node.

The same set of optimizations also proved useful for a risk management product that our company’s financial services practice was involved with.

The moral of the story is that rather than use parallel processing as a first choice given the abundance of cores and threads, it is much more effective (by orders of magnitude) to use common sense performance engineering and then use parallel processing as the last choice. We have demonstrated a 2000 fold single threaded increase in performance from 190 trades/sec to 400,000 trades/sec by a simple use of technical and business optimizations that appeal to common sense, which is really what a lot of performance engineering is about.

Acknowledgements
The authors would like to thank Mukesh Gangadhar from Intel Corporation for his tremendous support with optimizations around the Intel C compiler.

References


