LAB 49

INNOVATION ON DEMAND

Distributed Analysis with ScaleOut StateServer: Case Study

Prepared for

SCALEOUT SOFTWARE

By Lab49

April 6, 2009

Table of Contents

Summary

This report describes a study performed by Lab49 to investigate the benefits of using ScaleOut StateServer's distributed caching to boost the performance of real-world, data-parallel applications, particularly those specific to financial services. In focus is a new feature called parallel method invocation *(PMI), which stands to dramatically improve the performance of data-parallel computations while simplifying their overall structure. To better understand the potential of this new technology, Lab49 developed a representative financial services application using PMI to analyze data hosted in a distributed cache. Performance tests of this application were then conducted on a 32-node cluster at Microsoft's Enterprise Engineering Center. Test results demonstrated clear advantages for PMI in both performance and scalability over more traditional parallel computation techniques. Furthermore, in the process of developing the application, use of PMI was observed to greatly simplify transitions from sequential program execution to parallel program execution by shielding developers from complex distributed cache details. Given this, Lab49 expects that PMI's combined benefits will offer important advantages to developers of financial services applications.*

1. Challenge

The emergence of server farms and computational grids over the last decade has been driven by the need to provide scalable, highly available platforms for processing e-commerce, scientific, financial and other computationally-intensive tasks whose demands exceed the capabilities of one server. A major challenge for these grids is an ability to make computational inputs, such as financial market data, easily accessible for processing at runtime.

To date, the prevailing approach to this challenge has been to require logic on each node in a grid to retrieve its inputs from some remote storage, typically a back-end database. When computation on the node completes, intermediate results are copied back to remote storage in preparation for aggregation or further processing.

While effective in situations involving a relatively small number of light-weight inputs, this approach can exhibit non-scalable performance as the number of inputs grows or the inputs themselves become larger. In such cases, increased data motion between remote storage and computation nodes can cause bottlenecks in accessing storage and generate significant amounts of network traffic, hindering performance and grid scalability.

The relatively recent adoption of distributed caching has greatly alleviated the storage bottleneck issue. By hosting computation inputs and results in distributed cache memory, round trips to highly-contentious remote storage nodes are largely eliminated.

Despite this, however, problems persist. Data-parallel applications, such as financial calculations which analyze large sets of market data, can still place heavy burdens on cache networks as input data move from storage nodes to compute nodes at runtime.

The challenge remains as to how to most efficiently manage the placement of data within such caches so as to minimize network traffic and load-balance data-parallel computations for best overall performance and scalability. Coupled with this challenge is the need for a programming model that simplifies the structure of these computations and, in doing so, frees application developers from the burden of having to manage their complex details.

2. Solution

ScaleOut StateServer (SOSS) Grid Computing Edition seeks to address these challenges with the introduction of a new feature called *parallel method invocation* (PMI). On top of exposing a streamlined interface to simplify data-parallel application development, PMI provides a highly-optimized computational model for such applications. Building on the popular *map/reduce*[1] application structure, this model inverts traditional access for a distributed cache by automatically pushing computational logic to every node in the grid, thus eliminating the need to move cached data between compute nodes and storage nodes at runtime. Under PMI, node workers read their unique inputs from local memory rather than from remote databases or other cache servers.

3. Benefits

The key benefits to this new technology are dramatic increases in throughput and grid scalability, as well as lower development costs. Network traffic is all but eliminated as computational inputs for each node are retrieved locally. Furthermore, leveraging both multi-node and multi-core parallelism, nodes can be scaled out as computational loads increase to achieve real linear gains in throughput. Finally, complex cache interactions are encapsulated by the system, permitting developers on data-parallel projects to devote more time to business logic and much less time to details pertaining to cache infrastructure.

4. Analysis

Application Overview

To demonstrate these benefits, Lab49 designed and implemented a simple back-testing framework for automated equity trading strategies inspired by (Schoreels & Garibaldi, Genetic Algorithm Evolved Agent-Based Equity Trading Using Technical Analysis and the Capital Asset Pricing Model, 2006) and (Schoreels & Garibaldi, A Comparison of Adaptive and Static Agents in Equity Market Trading, 2005).

At a high level, the framework provides a common representation for trading strategies and a mechanism to measure the profitability of those strategies over many stock price histories. Specifically, it calculates the total daily P&L of applying a strategy to a portfolio that can only include cash and/or a single stock. Each portfolio is granted an initial cash position, and the strategy trades in and out of the stock over each day of its price history. The effectiveness of a strategy for any given stock is measured by summing the daily P&Ls at the stock's last closing price. The total effectiveness of a strategy across all stocks is measured by averaging those summed P&Ls.

Such a framework, though simplified here for the purposes of this study, is common in computational finance, both in academic research and commercial practice. The amount of market data required can be large (particularly when using intraday prices in addition to closing prices) and the time required to compute results across many strategies can be significant.

Performance and Development Considerations

Like most platforms in financial services, the effectiveness of the back-testing framework depends as much on throughput and performance as it does on program logic. Whether geared toward pricing, trading, or risk management, such systems are constantly required to achieve more in less time, outpacing competitors while minimizing business risk.

Traders and quantitative analysts rely on back-testing systems to guard against the risks of deploying newly-crafted automated trading strategies by simulating the outcomes of those strategies over potentially many years worth of market data for many different securities. System performance is critical to this workflow, as players strive to be

[1] For further reading on *map/reduce*, see (Jeffrey & Ghemaway).

first to market with successful trading innovations. Sluggish or unresponsive platforms requiring many hours or even days to produce results are deemed unacceptable.

While performance can, to some extent, be addressed at the program level, computations involving dozens of year's worth of price histories across thousands of securities and blending many different technical formulas can quickly overwhelm even the most powerful of commercially-available servers and dramatically increase system response times.

Compute grids offer a cost-effective means to overcoming these single-server limitations. By clustering multiple servers together on a network and joining them in common computational purpose, such demanding operations can be reduced to seconds or less, freeing system users to experiment with new strategies more rapidly and iteratively.

As mentioned above, however, these grids are not without pitfalls. With increasing computational loads come increasing levels of network traffic between grid servers. This traffic can quickly saturate network bandwidth, reducing or completely reversing anticipated performance gains.

Furthermore, due to the complex nature of grid-aware applications, implementation costs on such projects may grow steeply or even prohibitively, delaying development and potentially leading to missed market opportunities.

Enter PMI

SOSS/PMI aims to address these issues by providing a programming model that simplifies data-parallel implementations while simultaneously eliminating performance and scalability problems common to compute grids.

Ease of Development

Traditionally, application developers tasked with building systems that target grids have been forced to write code responsible for passing data inputs and results between grid nodes. Such logic, as noted previously, is often time consuming to implement and difficult to maintain.

PMI greatly improves this situation through its simplified interface. Per the terms of the system's *map/reduce* programming model, developers creating data-parallel applications need only implement two functions: an *eval* method to evaluate each one of a set of cached objects, and a *reduce* method to merge results from separate *eval* calls. Importantly, these two methods, which can be written in C#, Java, or C/C++, operate on in-memory objects without any knowledge of a distributed cache**.** This fact frees developers from having to deal with complex cache interactions. All necessary data migration to and from the cache is managed internally and seamlessly by SOSS/PMI.

Once the above implementations are in place, initializing parallel execution is trivial. Developers simply call a single *Invoke* method on the SOSS API, passing in delegates for their *eval* and *reduce* methods, as well as a query specification used to select cached objects for evaluation. All other details are handled by the system. SOSS automatically multicasts the query specification to all participating nodes in the grid. In parallel, these nodes execute that specification on their locally-cached objects, passing return values to a local engine which in turn executes *eval* using all available processors and cores. In-memory *eval* results are combined by calling *reduce* on the node to produce a single in-memory result. When all local computations have completed, SOSS merges these results to produce a final result object. This object is then returned to the *Invoke* caller.

At no point after having initially loaded data into the cache are developers forced to interface with the cache again. To illustrate the implications of this fact, it is helpful to recap the evolution of the strategy back-testing system described in this study. When building that system, Lab49 engineers began by coding a series of technical calculations to determine relative values of stock prices, and a simulation engine to drive those calculations. The simulation engine was designed to run sequentially, with no thought given at all to parallel execution of multiple engine instances. Only later in the process, when calculations and simulation internals had been finalized, was an effort made to run simulations in parallel across nodes in a compute grid. Following PMI interface guidelines, a single engineer with no prior PMI experience was able to successfully enable parallel simulation execution on a cluster of three test servers in a matter of hours. In the end, the required changes were deemed so light and unobtrusive that the programmatic option to execute sequentially (on a single server) or in parallel (on a cluster) was preserved as a command line switch. This switch was toggled on and off during testing, depending on cluster availability.

Performance Benefits

In addition to simplifying development of data-parallel applications, PMI also offers a means to resolving critically important compute grid performance issues.

As noted above, the prevailing approach to performing *map/reduce* over a grid is to place input data in remote storage, such as a database server or separate cache node, and then require workers on each compute node to copy their inputs from that remote storage during each map operation. Following each map operation, workers copy intermediate results back to remote storage in preparation for reduction, which is generally carried out by a single aggregation worker on a separate node. This approach, while fairly intuitive, can generate detrimental levels of network traffic and exhibit poor performance.

To overcome these problems, PMI effectively inverts the above approach. Mapping logic and subsets of input data are deployed side-by-side to all nodes in the grid. During map operations, inputs are sourced from local memory. Following map operations, intermediate results are written back to local memory. When all map operations complete, SOSS/PMI performs a fully distributed, parallel reduction across all cache nodes.

By taking advantage of a grid's statistically even distribution of objects across participating servers, PMI avoids the network overhead inherent in moving objects between nodes during computation, all but eliminating the possibility of network saturation. This, on top of the software's ability to make full use of each server's CPU cores at all stages of execution, combines to reduce overall program execution times to their bare minimums.

Test Plan

To concretely measure the performance benefits of this new technology, a test plan was formulated using, as its foundation, the strategy back-testing system described earlier.

Overview

Following this plan, approximately five years of historical price data were stored in SOSS across a variable number of grid servers for up to 4096 stock symbols gleaned from both the NASDAQ and NYSE. Price data was grouped by symbol into time series objects. Each time series contained approximately 1200 data points (one for each trade day in the five-year period under consideration), and each data point contained a trade date, a high price, a low price, a close price, an index close, and that day's prevailing risk free interest rate[2].

At runtime, all active nodes in the grid were called on to process time series data for a subset of symbols. For each symbol, the *eval* method (see above) was invoked, spinning up a trading simulation that accepted, as its computational input, a single time series object. In order to compare the relative performance of PMI versus random access to the distributed cache, two distinct techniques for retrieving those time series objects were

[2] For more information on the use of these fields, see Appendix A: Technical Indicators

applied and measured. In the first, the objects were read in automatically from local memory via the PMI engine. In the second, they were fetched from a separate, randomly selected node in the grid to imitate the effects of accessing inputs remotely prior to computation.

Once initialized in *eval*, each simulation iterated chronologically over the data points in its time series, passing them one-by-one to a set of eight different technical indicators[3]. These indicators in turn generated buy, sell or hold signals using their own internal policies. Signals were then merged to produce an overall weighted average signal, based on which mock trades were executed or no action was taken. Regardless of which, an end-of-day P&L was then calculated. When all data points in the time series had been processed, end-of-day P&Ls were summed to produce an overall P&L for the simulation. When all simulations had completed across all nodes in the grid, their results were merged via parallel execution of the *reduce* method (see above) to produce a P&L for the trading strategy as a whole.

Execution

The test plan was carried out over a three-day period at the Microsoft Enterprise Engineering Center in, Redmond, WA, on a cluster of 32 HP blades connected via Gigabit Ethernet. The blades were configured as follows:

- CPU: 3.0 GHz Intel Zeon, dual processor, 4 cores per processor
- Memory: 16 GB RAM
- Operating System: Windows HPC Server 2008
- Software framework: .NET 3.5

Per the plan, computation nodes were incrementally added in proportion to the number of time series objects under evaluation, beginning with four nodes/512 objects, and concluding with 32 nodes/4096 objects. At each four-node increment, runs were conducted using both PMI and random access to fetch time series inputs. Runs were repeated three times and average throughput rates were recorded.

Assuming an absence of bottlenecks to scalability, throughput rates were expected to grow linearly as nodes and time series objects were proportionally increased. Any drop off in throughput was taken to indicate the presence of at least one bottleneck, but primarily network saturation.

In order to gauge the effects of increasing input sizes on throughput rates, the entire sequence was repeated twice, once using fairly lightweight, 85-kb time series objects, and a second time using larger 2-mb time series objects. In total, 96 test runs were conducted.

Results

The results of these runs were compelling. Irrespective of time series object size, PMI decisively outclassed the random access approach both in terms of performance and scalability. Figure 1 below shows the results of the first sequence, which used 85-kb time series objects.

[3] For a complete description of the technical indicators used, see Appendix A: Technical Indicators

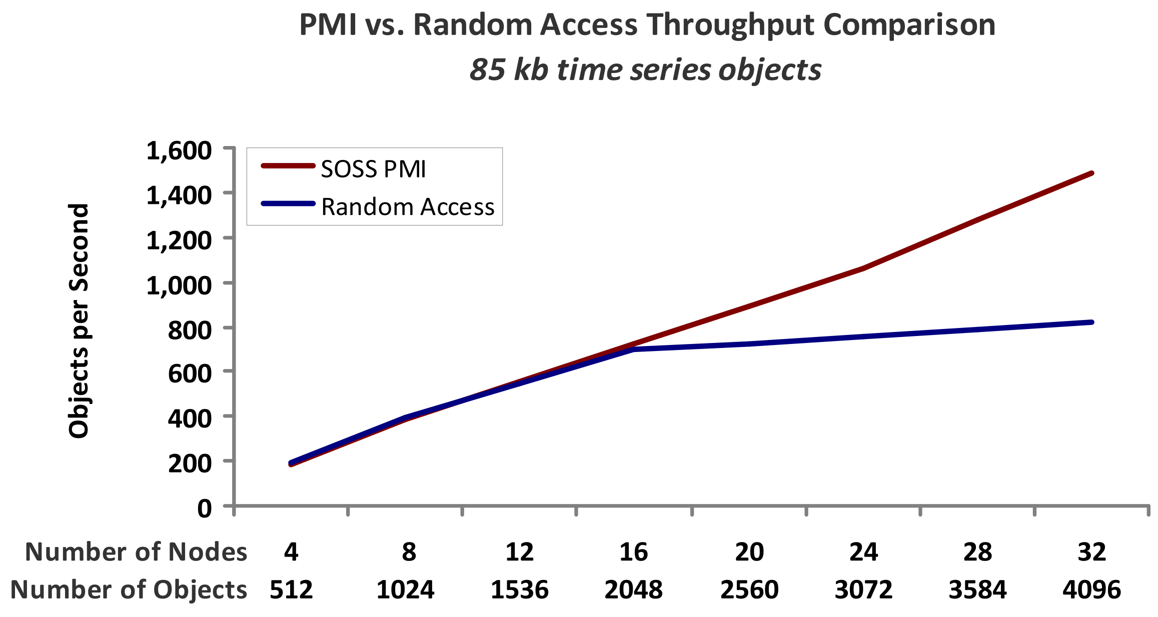

Figure 1: PMI vs. Random Access using 85-kb time series objects

Though random access demonstrated near linear throughput gains up to 16 nodes/2048 objects, throughput leveled off markedly beyond that point as traffic between nodes increased and the network subnet became saturated.

Conversely, PMI exhibited very linear gains at each step, beginning with four nodes/512 objects and continuing up through 32 nodes/4096 objects. At the 32-node point, PMI throughput was nearly 82% greater than that of random access. Seen purely in terms of execution times, PMI outpaced random access on this final run by a factor of more than 1.8, completing in 2.753 seconds versus random placement's 4.987 seconds.

To put these numbers into clearer context, sequential execution of 4096 objects on two cache nodes took more than 364 seconds to complete during a separate test run, lending even more weight to the case for compute grids and distributed caching in general. Though the 2.234 second difference between PMI and random access on the 4096 run may seem relatively insignificant in this light, the numbers behind this figure are in fact noteworthy. Using them to extrapolate test results involving 64 nodes/8192 objects, PMI is predicted to hold steady at around 2.753 seconds, while random access is seen increasing to roughly 7.031 seconds, more than 2.55 times slower. Taken to 128 nodes/16384 objects, random access increases to roughly 9.914 seconds, more than 3.6 times slower than PMI's essentially flat rate.

When time series object sizes were increased from 85 kb to 2 mb in the second sequence, the advantages of PMI over random access became even more pronounced. Figure 2 shows these results.

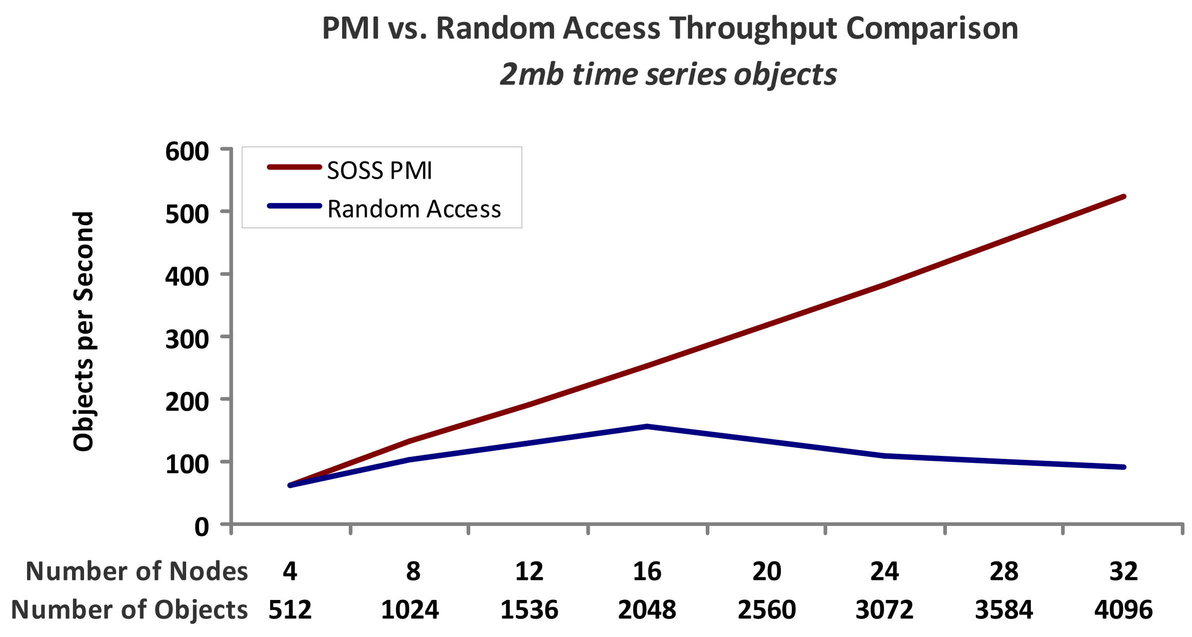

Figure 2: PMI vs. Random Access using 2-mb time series objects

Though random access throughput again increased in near-linear fashion up to 16 nodes/2048 objects, it did so at a significantly lower rate when compared to PMI. At the 16-node point, PMI throughput was more than 61% greater than that of random access.

Furthermore, while PMI for a second time demonstrated completely linear throughput gains at each step in the 2-mb test as nodes were scaled out and object counts increased, random access throughput peaked at 16 nodes/2048 objects and then steadily deteriorated as demand for increasing numbers of the larger computational inputs further saturated network bandwidth. At the 32-node point, PMI throughput was a massive 473% greater than that of random access. Put in runtime terms, PMI clocked in at 7.84 seconds on its final run, more than 5.7 times faster than random access at 44.953 seconds.

5. Conclusions

As these results demonstrate, SOSS/PMI offers clear advantages, both in terms of performance and scalability, over more traditional data management techniques common to compute grids today.

Though adequate in situations involving relatively few or small inputs, the approach of requiring each grid node worker to retrieve its data from remote storage at runtime can generate debilitating levels of network traffic. This traffic can potentially saturate network bandwidth and hinder if not completely prohibit throughput gains normally associated with scaling out a grid. Data-intensive applications, such as those common to computational finance, predictably suffer performance penalties under such circumstances.

SOSS/PMI offers a means to overcoming these problems. By evenly distributing both computational logic and input data to each node in a grid, the software effectively eliminates the network saturation issue. Workers on each node read computational inputs directly from local memory instead of remote storage, drastically reducing the costs of input retrieval. This method significantly improves system response times, particularly as inputs become more numerous or larger, and clears the way to achieving consistently linear gains in throughput as nodes in the grid are scaled out.

What's more, as development of the strategy back-testing system created for this study demonstrates, PMI's greatly simplified programming model places these parallel performance gains well within reach of even modestly-staffed development teams. By encapsulating data-parallel capabilities behind an easy-to-grasp interface, SOSS/PMI frees application developers from the hassles of managing complex inter-node communications, allowing them instead to focus more intently on the business problem at hand. This benefits organizations concretely by lowering system implementation and maintenance costs, and reducing platform time to market.

6. Appendix A: Technical Indicators

Eight technical indicators were selected for inclusion in the strategy back-testing system used to compare PMI and random access throughputs. Though the system does allow for indicators to be disabled or variably weighted, the eight were utilized on an equal-weighted basis across all test runs to maximize computational complexity.

The following are brief descriptions of each indicator and their respective uses within the system:

Simple Moving Average

A simple moving average (SMA) is the unweighted mean of some number of previous data points. For example, a 10-day simple moving average of closing prices is the mean of the closing prices of the previous 10 days.

Within the test system, SMAs are used to generate buy and sell signals when close prices cross above or below their 26-day averages. When a price moves above its 26-day SMA, the trend is considered bullish, and a buy signal is issued. Conversely, when a price moves below its 26-day SMA, the trend is considered bearish, and a sell signal is issued.

Exponential Moving Average

An exponential moving average (EMA) applies weighting factors to data points. The weighting for each previous data point decreases exponentially, lending more importance to recent observations without discarding older ones.

Within the test system, EMAs are used to generate buy and sell signals when close prices cross above or below their 26-day averages. When a price moves above its 26-day EMA, the trend is considered bullish, and a buy signal is issued. Conversely, when a price moves below its 26-day EMA, the trend is considered bearish, and a sell signal is issued.

Moving Average Convergence/Divergence

Moving Average Convergence/Divergence (MACD) builds on moving averages, which are lagging indicators, to create a momentum oscillator by subtracting a longer moving average from a shorter moving average. The resulting plot forms a line that oscillates above and below zero, without any upper or lower limits.

Within the test system, the "standard" MACD formula is used to form the plot line. This is the difference between a security's 26-day and 12-day exponential moving averages. A bullish MACD crossover occurs when the plot line moves above its own nine-day EMA, triggering a buy signal. Conversely, a bearish MACD crossover occurs when the plot line declines below its own nine-day EMA, triggering a sell signal.

Stochastic Oscillator

The Stochastic Oscillator is a momentum indicator that shows the location of the most recent close relative to the high/low range over a set number of periods. Closing levels that are consistently near the top of the range indicate buying pressure and those near the bottom of the range indicate selling pressure.

Within the test system, the oscillator is used to generate buy and sell signals for a security when it breaks below 20 or above 80. When the oscillator moves below 20, an asset is considered to be "oversold" and a buy signal is generated. Conversely, when the oscillator moves above 80, an asset is considered to be "overbought" and a sell signal is generated.

Bollinger Bands

Bollinger Bands are used to compare volatility and relative price levels over a period time. The indicator consists of three bands designed to encapsulate the major features of a security's price action. The three bands are:

1. A simple moving average (SMA) in the middle
2. An upper band (SMA plus 2 standard deviations)
3. A lower band (SMA minus 2 standard deviations)

Within the test system, these bands are used to generate buy and sell signals when prices break above the upper band or below the lower band. When a price moves below the lower band, an asset is considered to be "oversold" and a buy signal is generated. Conversely, when a price moves above the upper band, an asset is considered to be "overbought" and a sell signal is generated.

Relative Strength Index

The Relative Strength Index (RSI) compares the magnitude of a stock's recent gains to the magnitude of its recent losses and converts that information into a number ranging from 0 to 100.

Within the test system, RSI is used to generate buy and sell signals when the RSI of a given security breaks above 70 or below 30. If the RSI rises above 30, the security is considered to be "oversold" and a buy signal is issued. Conversely, if the RSI falls below 70, the security is considered to be "overbought" and a sell signal is issued.

Rate of Change

Rate of Change (ROC) is a very simple momentum oscillator that measures the percent change in price from one period to the next.

Within the test system, ROC is used to generate buy and sell signals when prices move above or below an 26-day ROC of zero. When ROC crosses zero from negative to positive territory, the trend is considered bullish and a buy signal is issued. Conversely, when ROC crosses zero from positive to negative territory, the trend is considered bearish and a sell signal is issued.

Capital Asset Pricing Model

The Capital Asset Pricing Model (CAPM) is a stock valuation tool that describes the relationship between the risk of a security, its market price, and its expected rate of return. CAPM states that the price of a stock is linked to two variables - the time value of money and the risk (or beta) of the stock itself.

Within the test system, CAPM is used to determine the expected rate of return for an asset. Once determined, that rate is then compared to the expected rate of return for the S&P 500 and to the prevailing rate of return on the 30-year US government bond. A security is considered worthy of holding if its CAPM-derived rate of return is greater than both the expected overall return rate of the S&P 500 and the return rate on the 30-year bond. In such cases, buy signals are issues. Conversely, an asset is deemed unworthy of holding if its CAPM-derived rate of return is less than either the expected overall return rate of the S&P 500 or the return rate on the 30-year bond. In such cases, sell signals are issued.

7. About ScaleOut Software

ScaleOut Software develops, licenses and supports scalable, highly available distributed caching software for server farms and compute grids. Their mission is to deliver technology that dramatically lowers costs for storing and processing workload data. With more than two-hundred customers worldwide, ScaleOut Software continues to evolve its production-proven technology to address customer needs in ecommerce, financial services, and other mission-critical applications.

The company recently released a new version of ScaleOut StateServer, their flagship distributed caching product. Called Grid Computing Edition, this version adds innovative support for grid computing by employing patent-pending technology to enable scalable and highly available map/reduce computations over cached data.

8. About Lab49

Lab49 is a technology consulting firm that builds advanced solutions for the financial services industry. We believe in small, expert engineering teams, working in close consultation with our customers. We have an excellent track record of successful project delivery in demanding environments. Lab49's agile, iterative methodology combines frequent delivery with fine-grained reporting, giving our clients a tremendous amount of control and visibility while allowing our teams to focus on achieving specific, valuable business goals.

Our clients include many of the world's largest investment banks, as well as top hedge funds and other industry participants (market data providers, technology companies, ECN's, etc). We have deep experience building systems across all asset classes, from equities to fixed income, derivatives, commodities, FX, hybrid instruments, and many types of complex structured products.

9. Works Cited

Dean, J. & Ghemaway, S., *MapReduce: Simplified Data Processing on Large Cluster.* Retrieved from http://labs.google.com/papers/mapreduce.html

Schoreels, C., & Garibaldi, J. M. (2005). A Comparison of Adaptive and Static Agents in Equity Market Trading. *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology* (pp. 393 - 399). Washington, DC, USA: IEEE Computer Society.

Schoreels, C., & Garibaldi, J. M. (2006). Genetic Algorithm Evolved Agent-Based Equity Trading Using Technical Analysis and the Capital Asset Pricing Model. Canterbury, UK.