ABSTRACT

Low-latency real-time option analytics feeds provide tick-by-tick implied volatilities and greeks based on exchange data. In order for the Black-Scholes implied volatility surface to exhibit the empirically observed skew or smile, a stochastic volatility model can be used to compute the option greeks. Because the European price under many stochastic volatility models only exists in semi-analytic form, frequent robust calibration of the model is computationally prohibitive. This paper explores three parallelization approaches for calibrating stochastic volatility models deployed on a multicore CPU cluster.

The contribution of this paper is to provide benchmarks demonstrating hybrid shared and distributed memory parallelization techniques using Python packages for robust calibration of stochastic volatility models. The focus here will be on the Heston and Bates models, but the results in this paper generalize to any of the exponential Levy models incorporating stochastic volatility and jumps and whose characteristic function can be expressed in closed form. We evaluated the performance for our implementation on a cluster of 32 dual socket Dell PowerEdge R410 nodes providing 256 cores in total. Using distributed memory parallelization, we obtain a speedup of up to 139x against the sequential version of the calibration error function evaluation and reduce the overall time taken to calibrate a chain of 1024 SPX options by a factor of 37x.

1. INTRODUCTION

A fundamental question for high frequency equity traders is how to measure the volatility of equity indices. Recently, the International Securities Exchange in partnership with Hanweck Associates launched a low-latency real-time option analytics feed which provides tick-by-tick implied volatilities and greeks from exchange data. This service marks a new era in high performance computational finance, one where the need for compute intensive option pricing is now at the tick-by-tick level. At the same time, practitioners also require the implied volatility surfaces generated by the option pricing model to exhibit a number of empirically observed key characteristics such as smiles or skews. For equity options, these features are the result of the 'leverage effect' which can be captured by mean reverting stochastic volatility models with spot return-volatility correlation. Examples of such models include the Heston model [5] and its extension to incorporate jumps which is referred to as the Bates Model [2]. Unfortunately, even for European options, these models are computationally intensive and difficult to robustly calibrate. The computational challenge is compounded as there is a need for frequent calibration during a short interval when processing high frequency data.

At the same time, there has been much interest by the quantitative finance community in leveraging the scientific computing software infrastructure provided through the open-source Python libraries scipy and numpy. A key concern shared by the community is how to effectively deploy python implemented financial computations in a parallel computing platform. This paper explores three enabling approaches for parallelization of the Heston Model calibration procedure, using Python with the multiprocessing and MPI packages deployed on a multicore CPU cluster. The first approach uses shared memory parallelism across each core of a single multicore node. The second approach uses distributed memory parallelism across multiple nodes in a cluster and finally the third approach uses both shared and distributed memory parallelism.

1.1 Literature review

The problem of how to effectively calibrate stochastic volatility models to quoted option prices is a topic of great interest to practitioner’s who require pricing models for building volatility surfaces to not only fit the market data but whose parameterization is stable over time. A primary reason for this is that the volatility surface is used to price more exotic options for which there are few if any market
quotes. Hence the mis-calibration of vanilla options can lead to significant pricing anomalies in exotic options. Heston’s stochastic volatility [5] has drawn widespread usage by its ability to capture the volatility smile and skew due to, for example, leverage effects. However the robust and fast calibration of this model requires judicious choice of numerical approximation and optimization schemes.

Mikhailov and Nögel [8] address the problem of calibrating Heston’s stochastic volatility model by providing guidance on how to calibrate the model to vanilla call options. The authors draw attention to the fact that the calibration procedure is non-trivial - it is a non-linear programming problem with a non-linear constraint and non-convex objective function. Since multiple local-minima may exist, Mikhailov et al. [8] propose using a combination of global search and local optimizers. The authors further note that the use of common stochastic algorithms for global search, such as simulated annealing, generally renders the calibration problem more computationally burdensome.

To improve the performance of this calibration approach, Aichinger, Binder, Fürst and Kletzmayer [7] implement a shared memory parallelization of the Heston model calibration routine on a multicore CPU SGI Altix 4700 and a GPU server with two C1060 cards and a GTX260 card. The authors compare the stability and performance of various off-the-shelf global optimizers before concluding that the best performance can be obtained by using a hybrid composed of one of a variety of global optimizers with a Levenberg-Marquardt unconstrained local optimizer. The global optimizers that the authors consider include the differential evolution (DE) algorithm and simulated annealing (SA), both of which have been employed elsewhere in the quantitative finance literature [1]. Finally, the stability of the hybrid calibration method is evaluated on a snapshot of the option chain on the FTSE-100 index.

The use of an unconstrained local optimizer is unsatisfactory for two reasons. First, bounds on the model parameters are necessary to ensure that the solution lies in a feasible region which is consistent with the modeling parameter definitions. Second, the non-linear constraint enforces the Feller condition which prevents the stochastic volatility process from reaching zero. Neither of these conditions are satisfied a priori by the use of the Levenberg-Marquardt optimizer and the latter condition is not satisfied by using an off-the-shelf global solver either. The authors address a violation of the Feller condition by a posteriori adjustment of the parameter set to satisfy the Feller condition. However, as the authors further note, this adjustment reduces the performance of the local solver.

In this paper, we consider the more general form of the Heston model, referred to as the Bates model [2], which includes jumps and is introduced in Section 2. Bates [2] notes that one of the reasons for adding jumps to a stochastic volatility model is to allow mixed maturity effects on skewness and excess kurtosis because the Heston model only permits skewness and excess kurtosis directly proportional to maturity. For European options pricing, the Bates models also permits a semi-analytic solution which is approximated by numerical methods, as described in Section 2.1.

Departing from Aichinger et al. [7], who consider calibration of just the Heston model, we deploy a DE global optimizer in combination with a local solver. However, we enforce the boundary conditions on the solver by using the truncated Newton constrained optimizer [9]. We further encode the Feller condition constraint by reparameterization of the parameter set. Details of the calibration are provided in Section 3.

The contribution of this paper is to evaluate the deployment of the Bates calibration algorithm on a cluster of multicore CPUs. In Section 4.1, we introduce three parallelization approaches based on shared memory, distributed memory and combined shared and distributed memory parallelism. Each of these approaches is evaluated in Section 5.

2. BATES MODEL

Following [2], recall that the Bates model is specified as the following set of coupled stochastic differential equations

\[
\begin{align*}
\frac{dS_t}{S_t} &= \mu dt + \sqrt{\theta} dW^{1}_t + (Y - 1)S_t dN_t, \\
\frac{dV_t}{V_t} &= \kappa (\theta - V_t) dt + \sigma \sqrt{V_t} dW^{2}_t,
\end{align*}
\]

describing the evolution of a stock price \( S_t \), whose variance \( V_t \) is given by a mean reverting square root process which ensures that the variance is always positive provided that \( 2\kappa\theta - \sigma^2 > 0 \). \( N_t \) is a standard Poisson process with intensity \( \lambda > 0 \) and \( Y \) is the log-normal jump size distribution with mean \( \mu_j = \ln(1 + a) - \frac{\sigma^2}{2} \), \( a > -1 \) and standard deviation \( \sigma_j \geq 0 \).

Both \( N(t) \) and \( Y \) are independent of the Wiener processes \( W^{1}_t \) and \( W^{2}_t \). A key characteristic of the model, which originates from the embedded Heston stochastic volatility diffusion model, is that the Wiener processes are correlated \( dW^{1}_t \cdot dW^{2}_t = \rho dt \). This feature enables the model to exhibit the leverage effect. Note that simply excluding the compound Poisson term \( (Y - 1)S_t dN_t \) recovers the Heston model.

With marginal loss of generality, we will restrict the scope of this section to European equity options. The Heston stochastic volatility model permits closed-form solutions for computing risk neutral European option prices. The price can be represented as a weighted sum of the delta of the European call option \( P_1 \) and \( P_2 \) - the probability that the asset price will exceed the strike price at maturity. Adopting standard option pricing notation, the call price of a vanilla European option is

\[
C(S_0, K, \tau; z_0) = S_0 P_1 - K \exp\left(-r \tau\right) P_2, \tag{3}
\]

\( P_1 \) and \( P_2 \) can be expressed as:

\[
P_j = \frac{1}{2} \int_0^\infty \left[ \frac{\exp\left(-iu\ln K\right)}{iu} \right] \frac{\phi_j(S_0, \tau, u; z_0)}{\phi_j(S_0, \tau, u; z_0)} du, j = 1, 2, \tag{4}
\]

where \( \phi_j \) are Bates analytic characteristic functions and are given in a convenient form in [6]. and \( z_0 \) is the vector of Bates model parameters\(^1\).

\(^1\)Note that throughout this paper, we use the property that the Bates characteristic function is just the product of
2.1 Numerical approximation

Under the Bates model, an option price can not be specified in closed form. Instead various numerical approaches have been widely adopted by practitioners. These include various quadrature schemes to approximate the integral in Equation 4. As described by Fang and Oosterlee [4], a popular choice is the Carr-Madan approach [3] which involves taking the Fourier transform of the risk neutral pricing formula with respect to the log-strike price to arrive at a closed form expression in Fourier space. A FFT algorithm can then be applied to the inverse Fourier integral. Fang et al. [4] note that quadrature methods are not the most efficient when solving Fourier transformed integrals as the integrands are highly oscillatory and require a fine grid for accurate resolution. Moreover, due to a singularity in the integrand, the accuracy of the Carr-Madan [3] approach depends on a regularization parameter, which from a practical perspective is undesirable.

To address these two issues, Fang et al. [4] propose reconstructing the entire inverse Fourier integral, rather than just the integrand, with its Fourier-cosine series expansion. By avoiding representation of a singular integrand, this Fourier-Cosine method is shown in most cases to exhibit exponential convergence rates and the computational complexity is linear. Under the Fourier-Cosine method, the Bates model call price is approximated as

\[ C(S_0, K, \tau; z_0) \approx K e^{-r\tau} \text{Re}\left\{ \sum_{k=0}^{N-1} \phi\left( \frac{k\pi}{b-a}; z_0 \right) \exp\left( \frac{ik\pi}{b-a} U_k \right) \right\}, \]  

(5)

where \( x := \ln(S_0/K) \) and \( \phi(w; z_0) \) denotes the Bates characteristic function of the log-asset price, \( U_k \) the payoff series coefficients and \( N \) denotes the number of terms in the cosine series expansion (typically 128 will suffice).

Figure 1 provides a comparison of the error convergence properties of the Fourier-Cosine, a fixed Gauss-Legendre quadrature and Carr-Madan FFT methods applied to the Heston pricing model. For simplicity, a Gauss-Legendre quadrature with second degree Legendre polynomials has been chosen. The y-axis measures the absolute error between the machine precision approximation of the option price \( V^* \) and \( V \), the approximation using \( N \) grid points or terms in the Fourier-cosine series, as shown by the x-axis. \( N \) corresponds to the number of characteristic function evaluations, which is the most computational complex part of the price function.

We observe that for a given level of error, the Fourier-Cosine method always requires a smaller number of terms than points in the Gauss-Legendre quadrature scheme. For example, for an error of \( 1.0 \times 10^{-5} \), the Fourier-Cosine method requires approximately 4x fewer terms than quadrature points. Moreover, any fixed quadrature scheme is prone to instabilities when the integrand is singular. The Carr-Madan FFT method, while robust to singularities, exhibits poor convergence rates - for an error of \( 1.0 \times 10^{-5} \), the FFT method requires approximately 160x more grid points than the number of series terms needed in the Fourier-Cosine method.

In order to arrive at a baseline serial implementation from the Heston characteristic function and the log-normal jump model, which to benchmark parallelization approaches, we ensure that we choose the numerical technique which is stable and exhibits the best error convergence properties. In doing so, we solve the characteristic function at a lower resolution without compromising accuracy in the option price and thus ensure that the serial implementation is advantageous with respect to reduced computational tasks resulting from superior numerical convergence properties.

3. CALIBRATION

Calibration of the Bates model involves finding the parameters which minimize the error between the Bates model prices and the observed prices across a set of options on the same underlying instrument, but whose contract maturities \( T \) and strikes \( K \) differ. This is formulated as a constrained non-linear least squares optimization problem of the form

\[ \min_{\theta} f(\theta) = \left( \sum_{i=1}^{K} \sum_{j=1}^{T} w_{ij} [V(S_0, K_i, \tau_j; \theta) - \hat{V}_{ij}]^2 \right)^{1/2}, \]

subject to the bound constraints \( a_i \leq z_i \leq b_i \) and the non-linear constraint \( 2\theta - \sigma^2 > 0 \) (the Feller condition), where the real parameter set \( \theta := [\theta, \sigma, \kappa, \rho, \nu_0] \). \( V(S_0, K_i, \tau_j; \theta) \) denotes the model option price and \( \hat{V}_{ij} \) denotes the quoted mid-price of the option with an underlying price \( S_0 \), maturity \( T \) and strike \( K_j \). The overall quality of fit is sensitive to the choice of weights. An intuitive choice is to emphasize the most liquid contracts in the chain by choosing the weights to be the reciprocal of the bid-ask spread \( w_{ij} = 1/(\hat{V}^{ask}_{ij} - \hat{V}^{bid}_{ij}) \).

In order to apply off-the-shelf global and local solvers to...
Equation 6 with a priori transformation of the Feller constraint, we apply the following transformation \( \kappa' = 2\kappa\theta - \sigma^2 \), with bounds \( \epsilon \leq \kappa' \leq b' \) where \( \epsilon << 1 \) and \( b' \) is given by twice the product of the upper bounds on \( \kappa \) and \( \theta \) minus the square of the lower bound on \( \sigma \). The parameter set in Equation 6 becomes \( \kappa' = [0, \sigma, \kappa', \rho, v_0] \) subject to \( a' \leq \kappa' \leq b' \). The original parameter \( \kappa \) is recovered from \( \kappa = \frac{\kappa' + \sigma^2}{2\theta} \) and since \( \kappa', \theta > 0 \) it follows trivially that \( \kappa > 0 \). We note however that we have surrendered direct control of the bounds of \( \kappa \), although in practice this can be accommodated provided there is flexibility on the choice of bounds of \( \sigma \) and \( \theta \). The other constraints are unaffected, \( \rho \in [-1, 1], v_0 > 0, a > -1, \sigma, > 0 \) and \( \lambda > 0 \).

4. IMPLEMENTATION

For calibrating the option price model we consider a sample chain of \( n \) option data \( \text{ch}[n] \), where the \( i \)th chain data has the following key information:

- \( \text{ch}[i].u \): Underlying asset price
- \( \text{ch}[i].s \): Strike price
- \( \text{ch}[i].m \): Time to maturity
- \( \text{ch}[i].p \): Option price\(^2\)

The calibration algorithm starts with an initial guess of five parameters and iteratively improves the guess using an optimization routine until it meets the convergence criteria. A typical organization of this computation involves calling an optimization routine with a pointer to the OptimizationAlgorithm until it meets the convergence criteria.

We combine the global optimizer with one of three constrained optimization solvers provided in the \texttt{scipy.optimize} package and callable through the \texttt{minimize} function. These optimizers are (1) the Sequential Least Squares Programming (LSQSP) method; (2) the L-BFGS-B algorithm; and (3) the Truncated Newton (TNC) method. Each method exploits the smoothness of the objective function over the feasible region by approximating the Jacobian with first order forward differences under perturbations of each parameter. A small number of Hessian vectors are also computed at each main iteration in the L-BFGS-B algorithm. The LSQSP method has the advantage that the change of parameterization detailed in Section 3 is not necessary as the method already incorporates non-linear inequality constraints.

The number of function evaluations per iteration is thus dependent on the number of model parameters. The global optimizer is terminated if either the objective function is below a threshold or the number of iterations exceeds a limit. The specifiable stopping criterion varies for each of the local optimizers. However, for ease of comparison of convergence properties between each, it is possible to terminate if either the absolute difference in function values between successive iterations is within a tolerance or the number of function evaluations exceeds a limit. In practice, a tolerance on the absolute difference of the function value is neither intuitive or ideally suited to calibration. In further experiments, not reported here, we find that specifying the tolerance on the norm of the difference in solution iterates leads to more stable parameters over successive calibrations. Of the three aforementioned local solvers, only the TNC method permits a tolerance of this form.

4.1 Parallel Implementation

For the parallel implementation, we explored three approaches using Python with \texttt{multiprocessing} and \texttt{mpi4py} packages. These are described by Algorithms 2-4. In the first approach, we just use the \texttt{multiprocessing} package to parallelize the option model computation (\( V \), line 3, Algorithm 1). For this we start \( np \) processes and assign a process to compute \( n/np \) of option price computations (see Algorithm 2). Essentially this parallelizes the loop of Algorithm 1 among \( np \) processes. The value of \( np \) is determined by the number of processors or cores available on the computer. Once a value of an option price is computed by a process it needs to be aggregated with values generated by other processes for computing the root mean square error (\( \text{rmsec} \)). This approach is very similar to OpenMP based parallelization and is suitable for shared memory architectures like a multi-core system.

The second approach just uses the \texttt{mpi4py} package and is suitable for a cluster of computers. In the MPI based approach, defined by Algorithm 3, we start \( np \) processes on a node and assign several option model computations to a pro-
cess. More specifically, we assign a chunk = n/mp of option model computations to a MPI process. A local rmsel value is generated by each process, see line 8 of Algorithm 3. The computation for rmse requires aggregation of local rmsel computed for each of the processes and is implemented using MPI collective computation primitives (line 10 of Algorithm 3).

Finally, the third approach is a hybrid approach that uses multiprocessing pools as well as MPI processes. The objective of implementing this approach was to explore the overheads of python multiprocessing package along with mpi4py package. In this approach, we start mp MPI processes, one on each node of the cluster. Next each process starts np processes, one for each core, using the multiprocessing pool (line 7 of Algorithm 4). The partial rmsel are generated at each node (line 12 of Algorithm 4) which are then summed using MPI allreduce to generate the global rmse (line 15 of Algorithm 4).

Algorithm 2 PARALLEL-ERRORFUNCTION(p)

1: \( \text{rmsel} \leftarrow 0 \)
2: Initialize: \( \text{pool} \leftarrow \text{Pool}() \)
3: \# create input parameters for all V computations
4: \( \text{inp} \leftarrow [([ch[i],u, ch[i], s, ch[i], m, p], 0) \leq i < n] \)
5: \# assign processes to computations
6: \( \text{res} \leftarrow [\text{pool.apply}(_{async}(V, ip), ip \in \text{inp})] \)
7: \( i \leftarrow 0 \)
8: for \( r \) in \( \text{res} \) do
9: \# get results from a process
10: \( \text{vp} \leftarrow r.get() \)
11: \( \text{diff} \leftarrow \text{ch}[i], p - \text{vp} \)
12: \( \text{rmsel} \leftarrow \text{rmsel} + \text{diff} \times \text{diff} \)
13: \( i \leftarrow i + 1 \)
14: end for
15: \( \text{rmsel} \leftarrow \text{SQRT}(\text{rmsel}/(m \times n)) \)
16: return \( \text{rmsel} \)

Algorithm 3 MPI-ERRORFUNCTION(p)

1: \( \text{rmsel} \leftarrow 0 \)
2: Initialize: \( i \leftarrow \text{GetMyRank()}, \text{chunk} \leftarrow n/mp \)
3: \( \text{rmse} \leftarrow 0 \)
4: for \( k = 0 \) to chunk − 1 do
5: \( \text{ko} \leftarrow \text{chunk} \times i + k \)
6: \( \text{vp} \leftarrow V[\text{ch[ko]}, u, \text{ch[ko]}, s, \text{ch[ko]}, m, p] \)
7: \( \text{diff} \leftarrow \text{ch[ko]}, p - \text{vp} \)
8: \( \text{rmse} \leftarrow \text{rmse} + \text{diff} \times \text{diff} \)
9: end for
10: \( \text{rmse} \leftarrow \text{ALLREDUCEMPI}(\text{rmse}) \)
11: \( \text{rmse} \leftarrow \text{SQRT}(\text{rmse}/n) \)
12: return \( \text{rmse} \)

5. EXPERIMENTAL RESULTS

We evaluated the performance for our implementation on a cluster of 32 dual socket Dell PowerEdge R410 nodes. The Dell PowerEdge R410 has two Intel Xeon E5504 processors with 4 cores on each processor for a total of 8 cores on a node. The sequential and parallel implementations were evaluated on one of the Intel Xeon E5504 processors of the cluster. We evaluated the parallel and hybrid approach on all the 32 nodes of the cluster.

An end-of-day snapshot of S&P 500 European options on August the 8th yields over 3000 mid-prices for use as a test set. To avoid calibrating to illiquid instruments, we exclude in-the-money calls and puts and further remove quotes with zero bids. This reduces the calibration set to approximately 1700 mid-prices. For convenience of benchmarking, we randomly select subsets of this chain in powers of 2 up to 1024 contracts. The short rate and dividend yield are set to 0.02% and 1.96% respectively. 128 terms are used in the Fourier-Cosine method for all benchmarking.

In our experiments we observed: (a) the overall time to calibrate Heston or Bates model is dominated by the ErrorFunction() routine both in sequential and parallel cases (more than 95%), and (b) the local solver is sensitive to the ordering of data in the RMSE calculation resulting in variation in number of iterations needed to converge for sequential and MPI based versions. For this reason, we report average execution time of a single iteration of ErrorFunction() in Table 1. The speedups numbers for different algorithms are based on average execution time of a single iteration of ErrorFunction() (Tables 2 and 3). To provide insight into the overall execution time, we also report the total calibration time in Table 5.

Table 1 compares the timing results in milliseconds of the Bates error function across the three approaches along with the speedups against the serial version shown in the parentheses. We observe that the timing of the sequential version scales linearly and that a chain of 1024 options costs 2.7s per error function evaluation.

Recall that Algorithm 2 is run by eight processes in a multiprocessing pool on a single node. Algorithm 3 is run by eight MPI processes on each node and Algorithm 4 is run by eight processes in a multiprocessing pool launched by a single MPI process on each node.

Overall for the range of problem sizes tested, we observe that the parallel efficiency of Algorithm 2 is the highest of the three parallel algorithms and that this increases with chain size. The parallel efficiency of Algorithm 4 is only compara-
Tables 2 and 3 show the speedup factors of the MPI and hybrid algorithms respectively. Each row in the tables represents a different chain size and each column represents the degree of parallelism. The results reinforce our existing observations - the MPI algorithm scales sub-linearly with the number of cores with parallel efficiency significantly increasing with problem size. On the other hand, we observe that parallel efficiency scales sub-linearly for the hybrid algorithm with the parallel efficiency increasing only marginally with problem size. We additionally observe for the smaller problem sets that the performance of the MPI algorithm does not monotonically increase with the number of cores. This property is due to inefficient degrees of parallelism in the data combined with the additional overhead of more processes. The performance of the hybrid algorithm, on the other hand, always increases with the number of cores.

Table 1: This table compares the elapsed wall-clock time of the error function in milliseconds between the three approaches. Speedups relative to the sequential version are shown in parentheses.

Table 2: This table shows the speedup factors for various numbers of cores against chain size using the MPI enabled algorithm. Each MPI process is mapped to a node with 8 cores.

Table 3: This table shows the speedup factors for various numbers of cores against chain size using the hybrid algorithm. Each MPI process is mapped to a node with 8 cores.

Table 4: This table shows the parameters of the global and local calibration routines used to produce the benchmark results shown in Table 5.

Table 5: This table compares the overall time (in seconds) to calibrate the Bates model across the local solvers and sequential and parallel implementations.
is significantly less efficient than the SLSQP and L-BFGS-B solvers, often requiring many more iterations to reach convergence. Moreover, the TNC solver is much more sensitive to the data and ordering of the RMSE evaluation, resulting in the MPI and Hybrid implementations exhibiting different convergence properties. This sensitivity explains the unexpected variations in timings between chain sizes and between the serial and MPI based implementations. Overall, the SLSQP method provides the best compromise between parallel scalability and robustness of calibration.

6. CONCLUSIONS
Low-latency real-time options analytics feeds provide tick-by-tick implied volatilities and greeks based on exchange data. In order for the implied volatility surface to exhibit the empirically observed leverage effect, jump-diffusion stochastic volatility models such as the Bates model can be used to calculate option greeks. Because the solution to many stochastic volatility models for European options only exists in semi-analytic form, frequent robust calibration of the model is computationally prohibitive. This paper explores three enabling approaches for the parallelization of stochastic volatility models deployed on a multicore CPU cluster. The performance of each parallel implementation is measured on a cluster of 32 dual socket Dell PowerEdge R410 nodes providing 256 cores in total. Using a distributed memory implementation, we obtain a speedup of up to 139x against the sequential version of the calibration error function evaluation. For an option chain of length 1024, we are able to reduce the entire calibration process time of the sequential implementation from 626s to around 17 seconds.

7. ACKNOWLEDGMENTS
The authors would like to gratefully acknowledge the support of Hanweck Associates and the International Securities Exchange in providing access to a data sample from the ISE/Hanweck Premium Hosted Database.

8. REFERENCES